

# HPS-3D160 面阵相机 SDK 使用手册



***HyperSen***  
HYPERSEN TECHNOLOGIES CO., LTD. 海伯森技术

## 目录

一、SDK 简介.....	4 -
二、将 SDK 集成到 IDE 中.....	4 -
2.1 Linux 平台下环境配置及集成到 IDE 中.....	4 -
2.1.1 HPS3D160 设备连接.....	4 -
2.1.2 工程环境配置与集成.....	5 -
2.1.3 在用户项目中使用 SDK.....	6 -
2.2 SDK Lite 集成到 IDE 中.....	9 -
2.2.1 添加头文件搜索路径.....	9 -
2.2.2 将 SDK 目录源码添加到项目中.....	11 -
2.2.3 移植到用户的平台.....	12 -
2.2.4 在用户项目中使用 SDK.....	12 -
2.2.5 使用 SDK 解析测量数据包.....	13 -
2.3 Windows 平台下将 SDK 集成到 IDE 中.....	14 -
2.3.1 添加 xxx.dll 动态链接库与 api.h 头文件到项目中.....	15 -
2.3.2 在用户项目中使用 SDK.....	15 -
2.4 ROS 平台下环境配置及将 SDK 集成到 IDE 中.....	19 -
2.4.1 创建一个工作空间.....	19 -
2.4.2 创建一个 ROS 包 (catkin 包).....	20 -
2.4.3 创建 ROS 消息 msg 和服务 srv.....	21 -
2.4.4 创建 ROS 的深度相机客户端节点和服务.....	24 -
三、API 函数接口.....	30 -
3.1 设置运行模式.....	30 -
3.2 获取/设置设备地址.....	30 -
3.3 获取设备版本信息.....	31 -
3.4 获取/设置数据包类型.....	31 -
3.5 保存/清除用户配置、恢复出厂设置.....	32 -
3.6 获取传输类型.....	32 -
3.7 选择 ROI 组/获取当前 ROI 组 ID.....	32 -
3.8 ROI 相关配置.....	32 -
3.9 获取当前设备支持的 ROI 数量和阈值数量.....	34 -
3.10 设置/获取输出/输入配置.....	34 -
3.11 设置 HDR 模式.....	35 -
3.12 设置/获取 HDR 配置.....	36 -
3.13 设置/获取距离滤波器配置.....	37 -
3.14 设置/获取平滑滤波器配置.....	38 -
3.15 设置/获取光学参数.....	38 -
3.16 设置/获取距离补偿.....	38 -
3.17 设置/获取畸变检测参数.....	39 -
3.18 设置/获取安装角度参数.....	39 -
3.19 设置/获取点云数据使能状态.....	40 -
3.20 将测量输出无效值设置为指定特殊值.....	40 -
3.21 设置边缘噪声滤除.....	40 -

3.22 保存点云数据为 ply 格式文件 .....	- 41 -
3.23 以太网版本连接服务器参数配置 .....	- 41 -
3.24 自动连接设备并进行初始化配置 .....	- 41 -
3.25 设置/获取测量数据包类型 .....	- 42 -
3.26 保存通信配置 .....	- 42 -
四、多设备支持 .....	- 42 -
五、常见问题 .....	- 43 -
4.1 编码格式不匹配? .....	- 43 -
4.2 如何获取点云数据和深度数据? .....	- 43 -
六、修订历史纪录 .....	- 44 -

## 一、SDK 简介

SDK 提供了 HPS3D160 面阵相机的应用程序接口, 目前可供 linux 平台, windows 平台, ROS 平台以及没有跑操作系统的大多数单片机上使用; 该 SDK 为二次开发包工具, 其提供的接口包含了我司生产研发的 HPS3D160 面阵相机的绝大部分操作指令, 请详细阅读该使用手册;

## 二、将 SDK 集成到 IDE 中

目前嵌入式程序的 IDE 环境大致可分为三种: Keil、IAR 和基于 Eclipse 的 IDE (例如 TrueStudio、Simplicity Studio 等), 每种的项目管理策略都不一致, 但是差异不是很大, 以下是不同平台下 SDK 的集成;

### 2.1 Linux 平台下环境配置及集成到 IDE 中

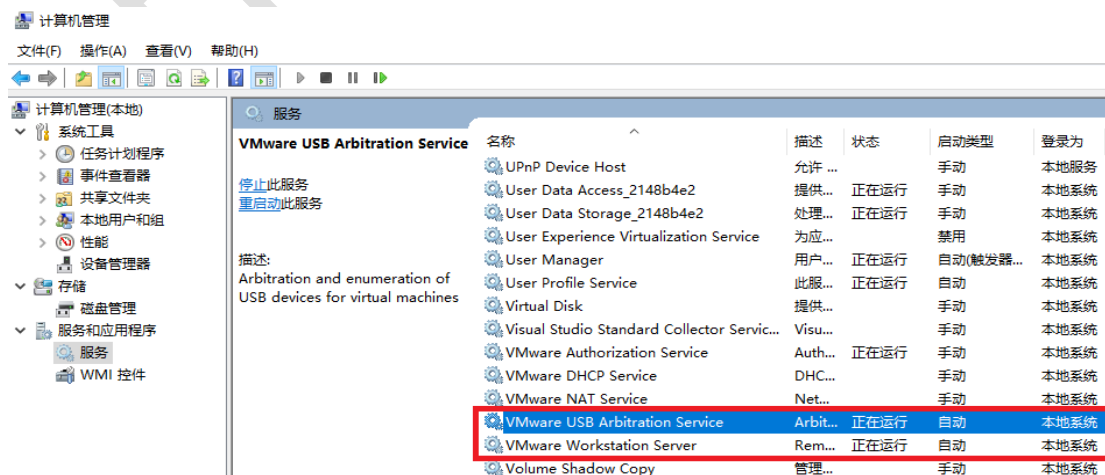
xxx.so 适合在 Linux 操作系统平台使用, 这里以 Ubuntu 为例。本示例基于 API 版本号 2018.12.04 V1.0.3 的 SDK 进行编写。

#### 2.1.1 HPS3D160 设备连接

将 HPS3D160 设备连接到电脑上, 打开终端输入 `ls /dev`, 查看设备 `ttyACM*`, 如下图所示:

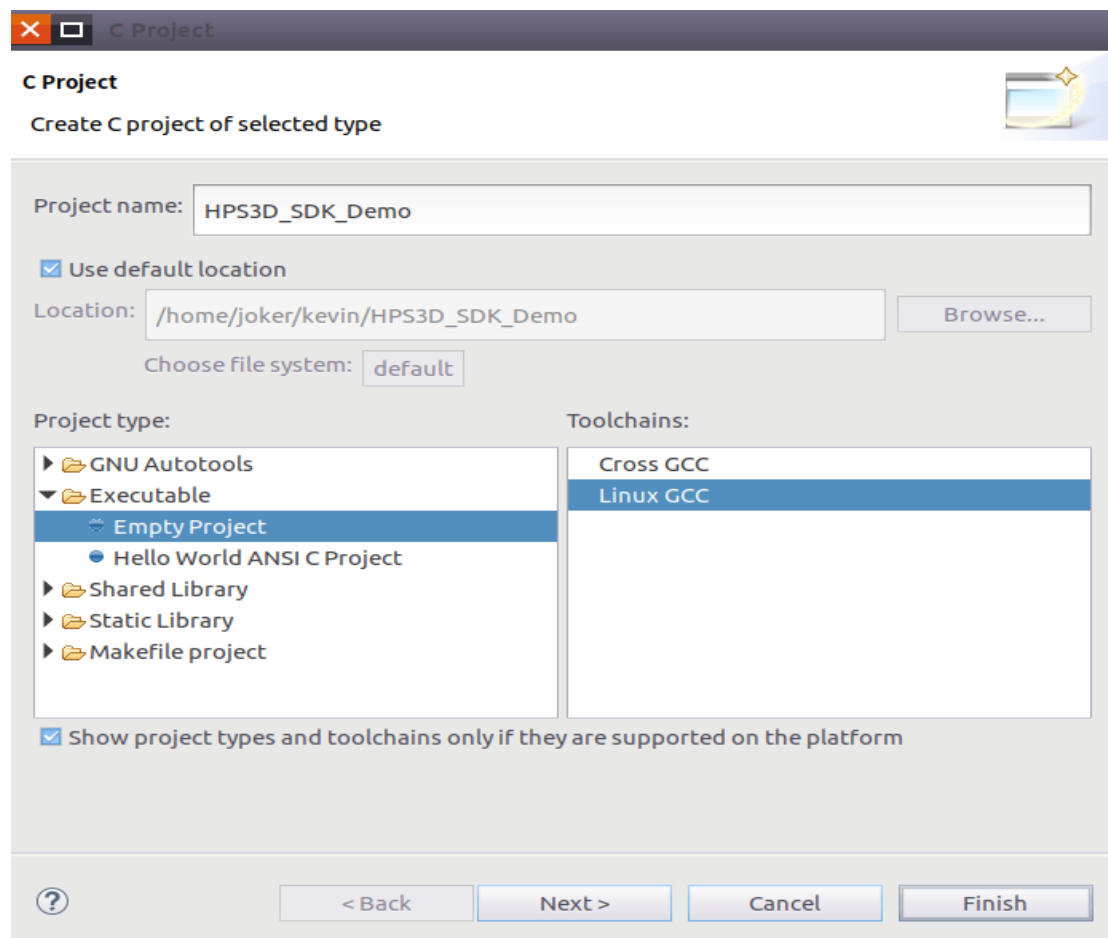
```
joker@Hypersen02:~$ ls /dev
agpgart      cpu_dna_latency  hpet      loop4      network_latency  sda      stdin      tty17      tty28      tty39      tty5      tty60
autofs       cuse             hugepages loop5      network_throughput sda1     stdout     tty18      tty29      tty4      tty50      tty61
block        disk             hwrng     loop6      null            sda2     tty       tty19      tty3      tty40      tty51      tty62
bsg          dmideid         initctl   loop7      port            sda5     tty0      tty2       tty30      tty41      tty52      tty63
btrfs-control dri              input     loop-control ppp          serial   tty1      tty20      tty31      tty42      tty53      tty7
bus          dvd             kmsg      mapper      psaux         sg0      tty10     tty21      tty32      tty43      tty54      tty8
cdrom        ecryptfs        lightnvm  mcelog     ptmx          sg1      tty11     tty22      tty33      tty44      tty55      tty9
cdwr         fb0             fd        loop0      random         snapshot tty12     tty23      tty34      tty45      tty56      ttyACM0
char         full            fuse      loop1      rkill         snd       tty13     tty24      tty35      tty46      tty57      ttyprintk
console      hidraw0         loop2     mtd        rtc           sr0      tty14     tty25      tty36      tty47      tty58      ttyS0
core         loop3          net       rtc0       stderr         tty15     tty26      tty37      tty48      tty59      ttyS1
cpu          loop4          network_latency sda      stdin      tty17      tty28      tty39      tty5      tty60
```

若没有查看到 `ttyACM*` 设备名称, 则需要重新拔插, 再次查看, 若还是没有, 则到“计算机管理->服务和应用程序->服务”中查看 VMware USB Arbitration Service 是否正在运行, 若禁用, 则开启运行, 之后重新拔插设备即可; 若不想每次登陆虚拟机都启动一次 USB 设备服务, 则可将 VMware Workstation Server 和 VMware USB Arbitration Service 均设置成运行和自动, 重启计算机即可。

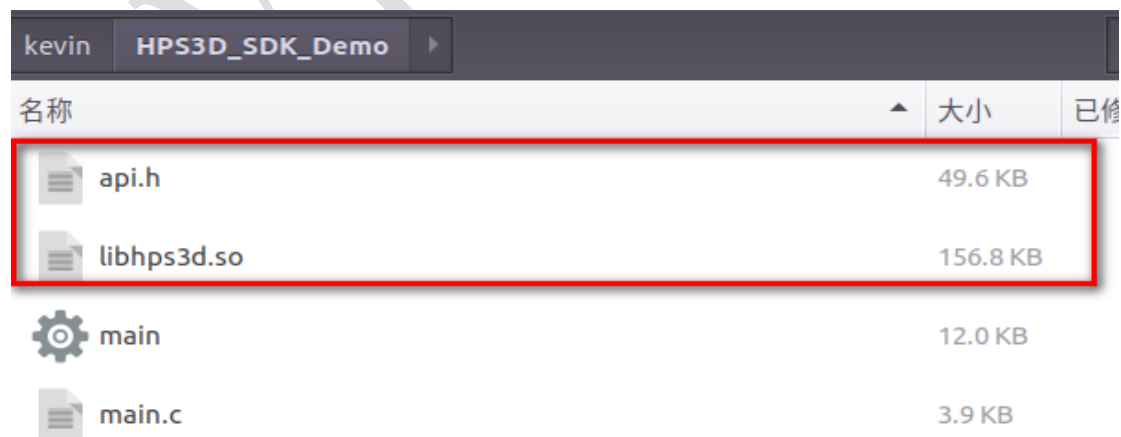


## 2.1.2 工程环境配置与集成

在 ubuntu 下的 eclipse 建立工程（其他 IDE 工具均可用），这里以 eclipse 为例，如下配置完后，点击 finish 完成工程搭建。



将 api.h 和 libhps3d.so 文件拷贝进工程目录；



方法一：

打开终端，输入 `sudo cp libhps3d.so /usr/local/lib/`，将 libhps3d.so 拷贝到 /usr/local/lib/ 目录下；之后执行 `sudo ldconfig` 进行加载。

```
@Hypersen02:~/kevin/HPS3D_SDK_Demo$ sudo cp libhps3d.so /usr/local/lib/  
@Hypersen02:~/kevin/HPS3D_SDK_Demo$ sudo ldconfig
```

方法二：在编译时加入指定动态库链接地址如 `gcc main.c -Wl,-rpath=./ -L./ -lhps3d64 -o app`

在 main.c 写入测试代码后，在终端输入：gcc main.c -L./ -lhps3d -o app，进行编译连接，再使用 sudo ./app 执行程序

```
@Hypersen02:~/kevin/HPS3D_SDK_Demo$  
@Hypersen02:~/kevin/HPS3D_SDK_Demo$ gcc main.c -L./ -lhps3d -o app  
@Hypersen02:~/kevin/HPS3D_SDK_Demo$  
@Hypersen02:~/kevin/HPS3D_SDK_Demo$  
@Hypersen02:~/kevin/HPS3D_SDK_Demo$  
@Hypersen02:~/kevin/HPS3D_SDK_Demo$ sudo ./app
```

选择可连接的设备，初始化成功后即可正常输出测量结果：

```
当前可连接的设备(请选择):  
0: /dev/ttyACM0  
请输入对应的序号:  
0  
初始化成功  
测量平均距离值为1706  
测量平均距离值为1706  
测量平均距离值为1734  
测量平均距离值为1715  
测量平均距离值为1689  
测量平均距离值为1705  
测量平均距离值为1724  
测量平均距离值为1700  
测量平均距离值为1693  
测量平均距离值为1707
```

### 2.1.3 在用户项目中使用 SDK

1、在项目中包含头文件，示例代码：

```
#include "api.h"
```

2、调用设备连接函数，USB 版本在此之前需要修改 handle->DeviceName，此参数为设备名完整路径及名称；以太网版本需先输入服务器 ip 地址及端口号，设置完成后执行如下操作：

```
HPS3D_HandleTypeDef handle;  
RET_StatusTypeDef ret = RET_OK;  
ret = HPS3D_Connect(&handle);  
if(RET_OK != ret)  
{  
    printf("设备打开失败! ret = %d\n", ret);  
    break;  
}
```

3、调用初始化配置函数，此步骤必须在步骤 2 成功执行之后操作。示例代码：

```
/*设备初始化*/  
ret = HPS3D_ConfigInit(&handle);  
if(RET_OK != ret)  
{  
    printf("初始化失败:%d\n", ret);  
    break;  
}  
printf("初始化成功\n");
```

**注：**

(1) 此步骤执行完成后将会自动获取当前设备的设备地址也是帧 ID，此参数默认保存在

handle->DeviceAddr 中。

(2) 请务必检查初始化函数的返回值，根据返回值判断是否初始化成功。SDK 的绝大多数 API 都提供了操作状态返回，建议用户对每个 API 的返回值进行检查，以确保可靠。

4、根据用户需要调用其他命令函数接口，对相机进行配置，配置完成后设置运行模式。注其他参数配置必须在此函数执行之前完成，否则配置将无法生效，示例代码：

```
/*将运行模式设置成连续模式*/  
handle.RunMode = RUN_CONTINUOUS;  
HPS3D_SetRunMode(&handle);
```

**注：**

(1) 设备通信为同步模式，执行任意命令函数时均会发送停止测量命令。所以在连续测量过程中，调用命令函数接口后，需要重新设置运行模式。

(2) 在运行模式配置，可选择三种模式：

①RUN\_IDLE：空闲模式，在此模式下，相机进入待机模式。

②RUN\_SINGLE\_SHOT：单次测量模式，在此模式下，相机进行一次测量，然后自动切换到 RUN\_IDLE。有两种测量方式，一种是同步测量，一种是异步测量（默认是异步）。

③RUN\_CONTINUOUS：连续测量模式，在此模式下，相机将自动进行连续测量，直到用户手动设定为 RUN\_IDLE 或 RUN\_SINGLE\_SHOT 模式；测量数据通过结构体形式返回，参考 MeasureDataTypeDef 结构体，数据类型通过返回包类型枚举(RetPacketTypedef)进行区分。

5、在所有步骤配置成功后，有两个方式获得测量数据，一种是同步测量方式（只支持单次测量），另一种是异步方式（支持单次测量和连续测量）。

(1) 同步测量（只支持单次测量），调用 api.h 中的 HPS3D\_SingleMeasurement 函数进行同步单次测量。示例代码如下：

```
HPS3D_SingleMeasurement(&handle);  
printf("测量平均距离值为%d\n", handle.MeasureData.full_depth_data->distance_average);
```

**注：**在 HPS3D\_SingleMeasurement 函数内将通信模式 handle.SyncMode 设成同步模式 SYNC，如需设置成异步模式，则在函数后将 handle.SyncMode 设成异步模式 ASYNC。

(2) 异步测量（支持单次测量和连续测量）

①设置运行模式，代码如下：

```
/*设置运行模式为连续模式*/  
handle.RunMode = RUN_CONTINUOUS;  
/*设置运行模式为单次模式*/  
handle.RunMode = RUN_SINGLE_SHOT;  
HPS3D_SetRunMode(&handle);
```

②编写观察者回调函数，代码如下：

```
/*观察者回调函数*/  
void * User_Func(HPS3D_HandleTypedef *handle, AsyncIOObserver_t *event)  
{  
    if(event->AsyncEvent == ISubject_Event_DataRecvd)  
    {  
        switch(event->RetPacketType)  
        {  
            case SIMPLE_ROI_PACKET:  
                break;  
            case FULL_ROI_PACKET:
```



```

        break;
    case FULL_DEPTH_PACKET:
        printf("测量平均距离值为%d\n", event->MeasureData.full_depth_data->distance_average);
        break;
    case SIMPLE_DEPTH_PACKET:
        break;
    case NULL_PACKET:
        break;
    default:
        printf("系统错误\n");
        break;
    }
}

return 0;
}

```

③观察者初始化，代码如下：

```

/*观察者回调函数与初始化*/
AsyncIOObserver_t My_Observer; /*定义观察者*/
My_Observer.AsyncEvent = ISubject_Event_DataRecvd; /*观察者订阅事件为数据接收事件*/
My_Observer.NotifyEnable = true; /*使能观察者*/
My_Observer.ObserverID = 2; /*设置观察者 id，目前仅支持单一观察者*/

```

④添加异步观察者（注册事件），代码如下：

```

/*添加异步观察者*/
HPS3D_AddObserver(&User_Func, &handle, &My_Observer); /*添加观察者，注册通知事件*/

```

6、步骤 1-5 的配置就可以正常测量数据，默认得到的数据是完整的深度图数据包（含深度数据）。返回的数据包类型有四种：简单 ROI 数据包（不含深度数据）、完整 ROI 数据包（含深度数据）、简单深度数据包（不含深度数据）、完整深度数据包（含深度数据）。还可以设置转换成点云数据包（只有完整 ROI 数据包和完整深度数据包才能进行点云格式的转换）。

**注：深度数据存储在一维数组，按顺序存放，若需要自行遍历。**

（1）默认是完整的深度图数据包（含深度数据）输出，若要配置成简单数据包输出，则需调用函数 HPS3D\_SetPacketType，进行设置。代码如下：

```

/*设置数据包类型*/
handle.PacketType = PACKET_SIMPLE; /*设置成简单数据包*/
HPS3D_SetPacketType(&handle);

```

（2）若是要配置成 ROI 数据包（简单和完整包同（1）配置），只需设定好 ROI 区域，并使能 ROI 即可，若需设置阈值自行配置，代码如下：

```

ROIConfTypeDef roi_conf;
/*设定 ROI 区域*/
roi_conf.roi_id = 0;
roi_conf.left_top_x = 10;
roi_conf.left_top_y = 10;
roi_conf.right_bottom_x = 30;
roi_conf.right_bottom_y = 20;

```



```
HPS3D_SetROIRegion(&handle, roi_conf);  
HPS3D_SetROIEnable(&handle, roi_conf. roi_id, true);
```

(3) 若要配置点云数据输出,则需要**在设置运行模式之前**,调用光学参数使能函数 HPS3D\_SetOpticalEnable 和使能点云数据输出的函数 HPS3D\_SetPointCloudEn,在单次测量和连续测量下返回的数据就是点云数据,代码如下:

```
HPS3D_SetOpticalEnable(&handle, true);  
HPS3D_SetPointCloudEn(true);
```

本 SDK 得到的点云数据是有序点云数据,点云数据格式是使用 (x,y,z) 空间坐标作为点云数据;提供结构体(在 api.h 中),代码如下:

```
/*每点的点云数据*/  
typedef struct  
{  
    float32_t x; /*x, y, z 空间坐标*/  
    float32_t y;  
    float32_t z;  
}PerPointCloudDataTypeDef;  
/*有序点云数据*/  
typedef struct  
{  
    PerPointCloudDataTypeDef point_data[MAX_PIX_NUM]; /*点云坐标 */  
    uint16_t width; /*宽度: 一行, 点的数目*/  
    uint16_t height; /*高度: 行的总数*/  
    uint32_t points; /*点云图, 点的总数*/  
}PointCloudDataTypeDef;
```

#### 注:

- (1) 在使能输出点云数据前需要开启光学使能,目的:为了得到垂直距离。
- (2) 在深度图的数据中,有**无效点**,在这里处理无效点也给出空间坐标,空间坐标(x,y,z)值为: z=distance[] (保留原有的无效数据值); x,y 则是体现了 z 在 distance 中的位置(即无效点的位置)
- (3) 在返回数据包结构体 MeasureDataTypeDef 类型中,定义的点云数据包是结构体数组 PointCloudDataTypeDef,对于深度图数据转换的点云数据存储在数组的[0]中,对于 ROI 数据转换的点云数据则在数组按顺序存储。

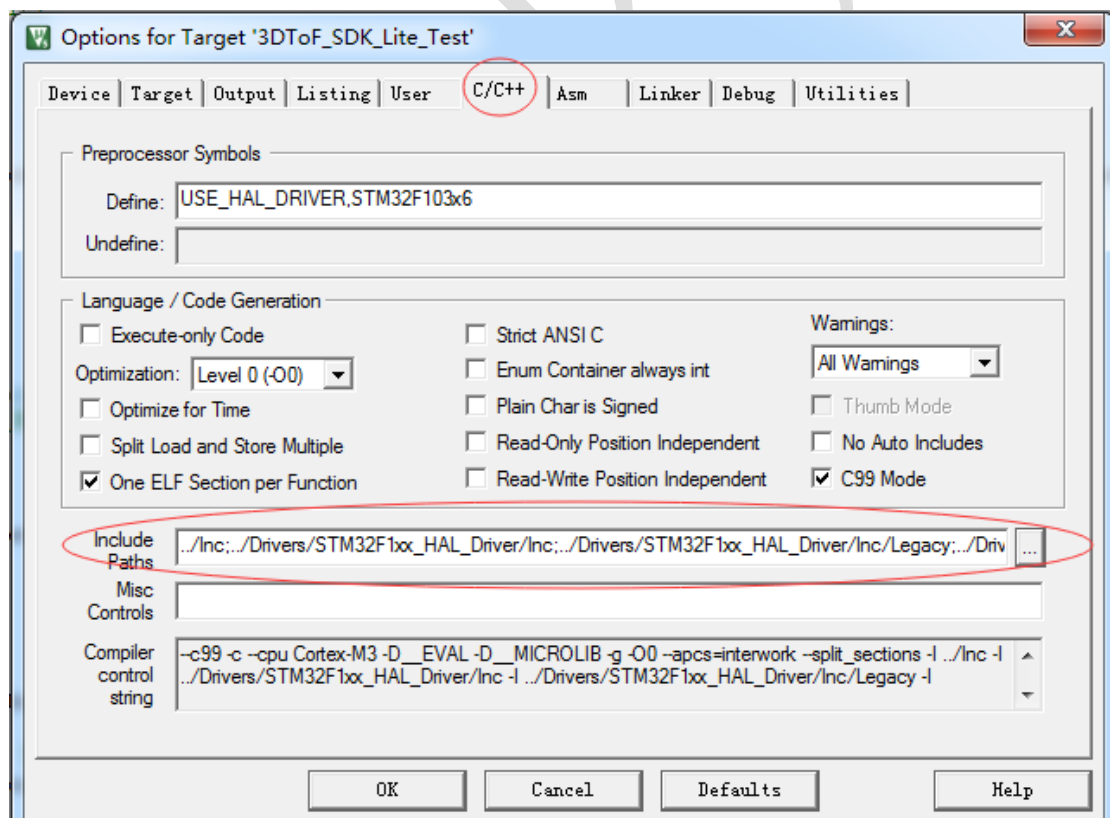
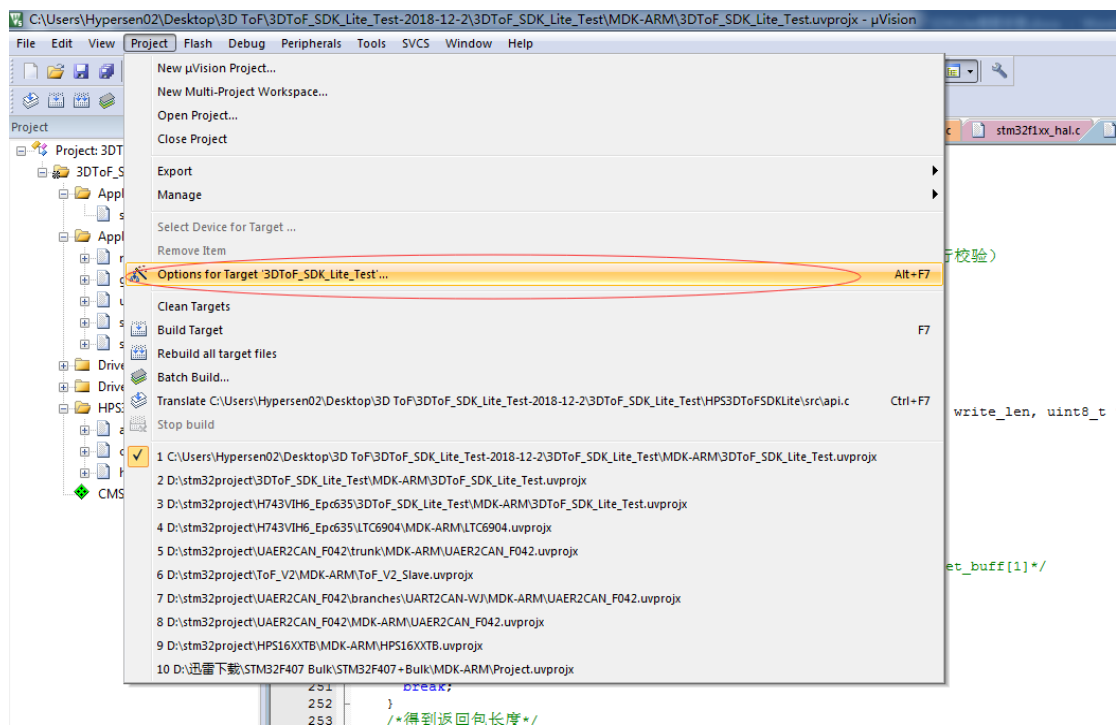
## 2.2 SDK Lite 集成到 IDE 中

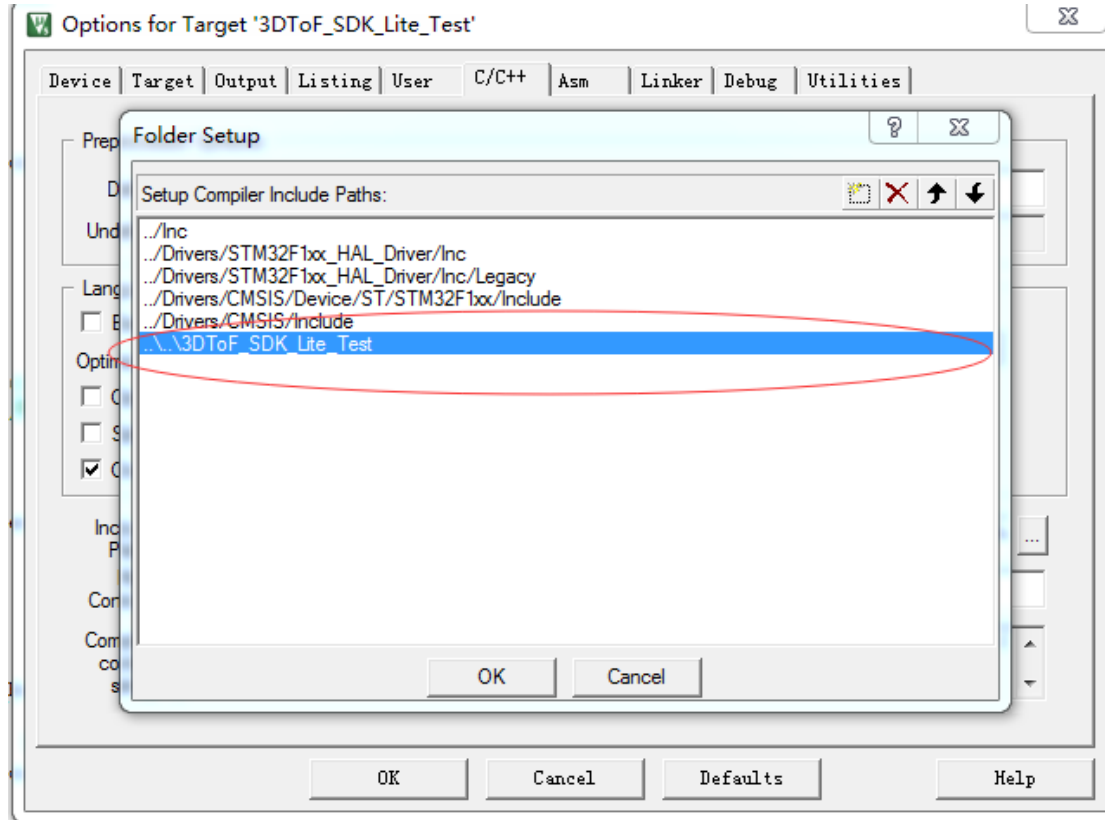
SDK Lite 提供了 HPS3D160 深度相机的轻量级应用程序接口,适合没有跑操作系统的大多数单片机平台使用,鉴于各种 MCU 平台的差异性,提供了源码级的 SDK,因此需要集成进用户的项目源码进行编译。SDK Lite 为轻量级二次开发包,仅提供基础的操作接口,由于深度图和完整的 ROI(敏感区域)数据占用内存较大(可能需要 1Mbyte 以上的 RAM),所以 SDK Lite 目前不支持深度图数据和完整的 ROI 数据,仅支持精简的数据包格式解析;该示例基于 SDK 版本号为 2018.12.03 V1.0.0 的 SDK 进行编写。

### 2.2.1 添加头文件搜索路径

SDK Lite 中的头文件包含的路径使用的是相对于项目根目录的相对路径,要保证 SDK 能

够正常编译，需要在 IDE 中将项目根目录路径包含进头文件搜索路径中，以 Keil 为例：

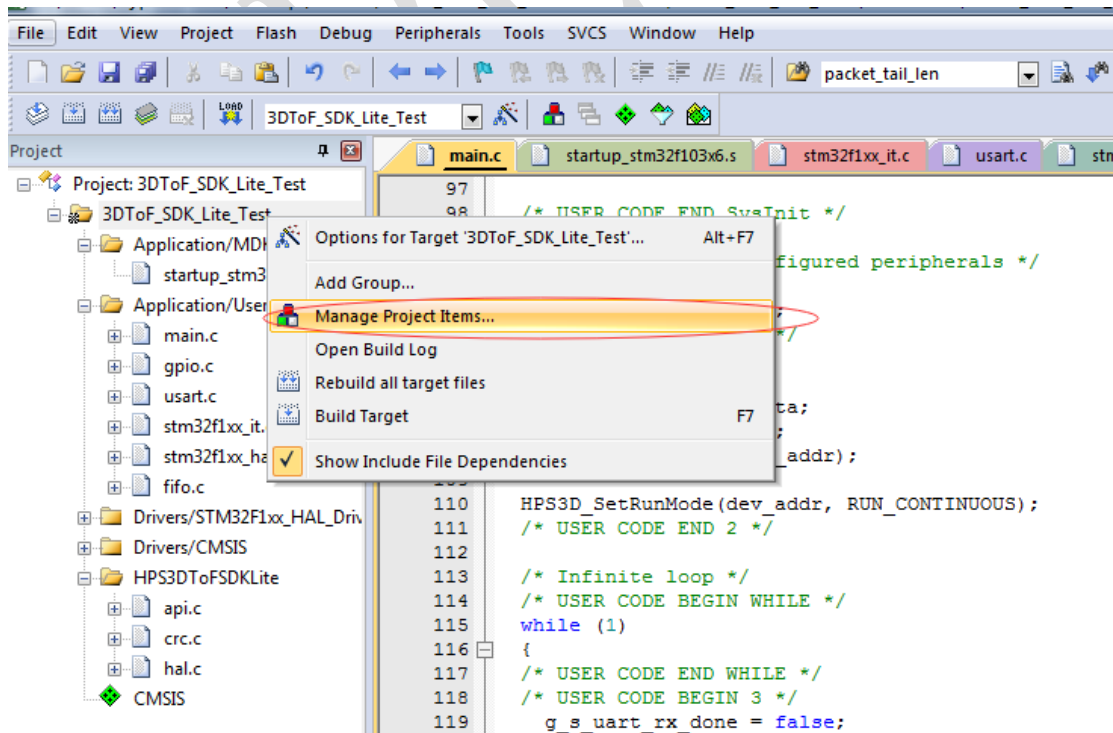


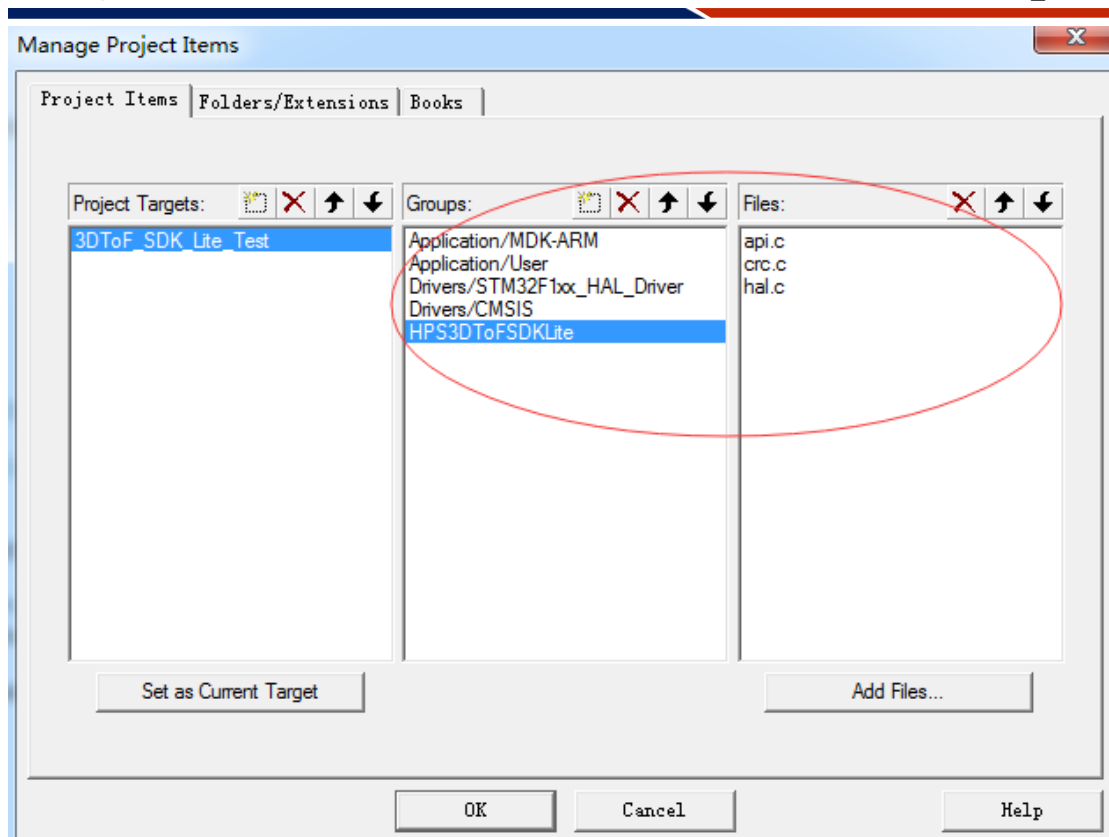


图上 3DToF\_SDK\_Lite\_Test 为项目的根目录。

## 2.2.2 将 SDK 目录源码添加到项目中

以 Keil 为例，将 SDK 目录源文件添加到项目中：





### 2.2.3 移植到用户的平台

SDK Lite 仅支持 RS232 和 RS485 接口的 HPS3D160 深度相机，由于平台的差异性，需要对通讯接口等底层进行移植。

- 1、编辑 **HPS3DToFSDKLite/src/hal.c** 文件
- 2、适配 Uart\_Read 和 Uart\_Write 等接口即可，以基于 HAL 库的 STM32 平台为例：

```
void Uart_Read(uint8_t *dest_buff, uint16_t length, uint32_t timeout_ms)
{
    HAL_UART_Receive(&huart1, dest_buff, length, timeout_ms);
}

void Uart_Write(uint8_t *from_buff, uint16_t length, uint32_t timeout_ms)
{
    HAL_UART_Transmit(&huart1, from_buff, length, timeout_ms);
}

void Delay_Ms(uint16_t ms)
{
    HAL_Delay(ms);
}
```

### 2.2.4 在用户项目中使用 SDK

1. 在源文件中包含头文件，示例代码：  
`#include "HPS3DToFSDKLite/inc/api.h"`
2. 调用初始化 API 对相机进行相应初始化，该函数会将相机的数据包格式设定为精简格式，并停止当前的连续测量，返回当前的相机的设备地址，这个地址在其他 API 中会使用到，请务必检查初始化函数的返回值，根据返回值判断是否初始化成

功。SDK 的绝大多数 API 都提供了操作状态返回，建议用户对每个 API 的返回值进行检查，以确保可靠。示例代码：

```
HPS3D_Initialize(&dev_addr);
```

3. 通过步骤 2 后便可以使用 SDK 中的 API 对数据包进行解析以及配置相机的参数。

### 2.2.5 使用 SDK 解析测量数据包

SDK 提供了数据解析 API，但不实现测量数据的接收，因为这与平台相关性太强。

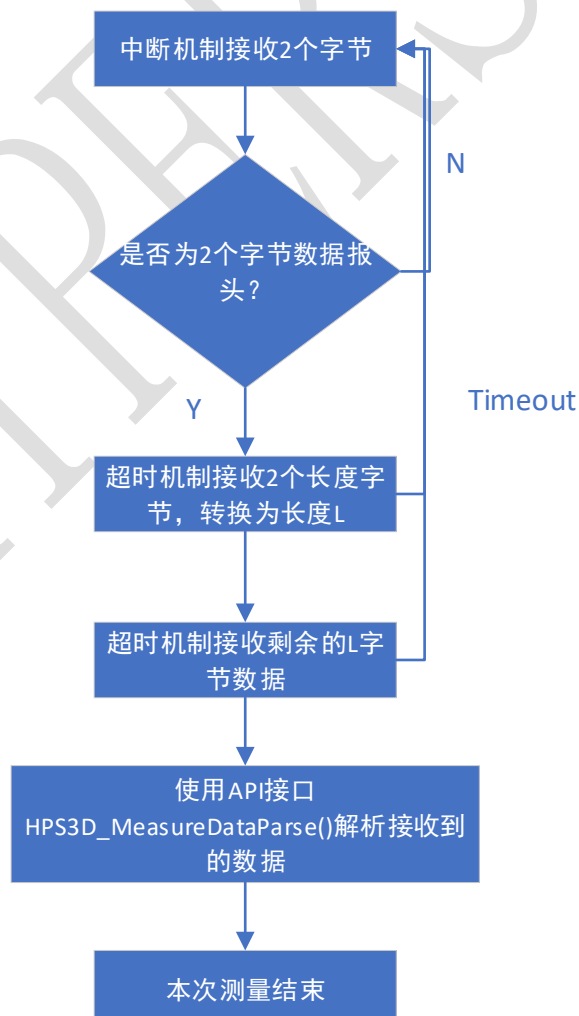
1. 设定相机的运行模式，可选三种模式：

- 1) **RUN\_IDLE**:空闲模式，在此模式下，相机进入待机模式，不进行任何测量。
- 2) **RUN\_SINGLE\_SHOT**:单次测量模式，设定此模式后，相机进行一次测量，然后自动切换到 **RUN\_IDLE**。
- 3) **RUN\_CONTINUOUS**:连续测量模式，设定此模式后，相机将自动进行连续测量，直到用户手动设定为 **RUN\_IDLE** 或 **RUN\_SINGLE\_SHOT** 模式，在此模式下，数据会通过 **RS485** 或 **RS232** 连续输出测量数据。

2. 串口接收测量数据包，数据包格式请参考 HPS-3D160 规格书，请按照格式定义接收数据包，可参考以下的数据接收流程结合自身平台进行相应改进：

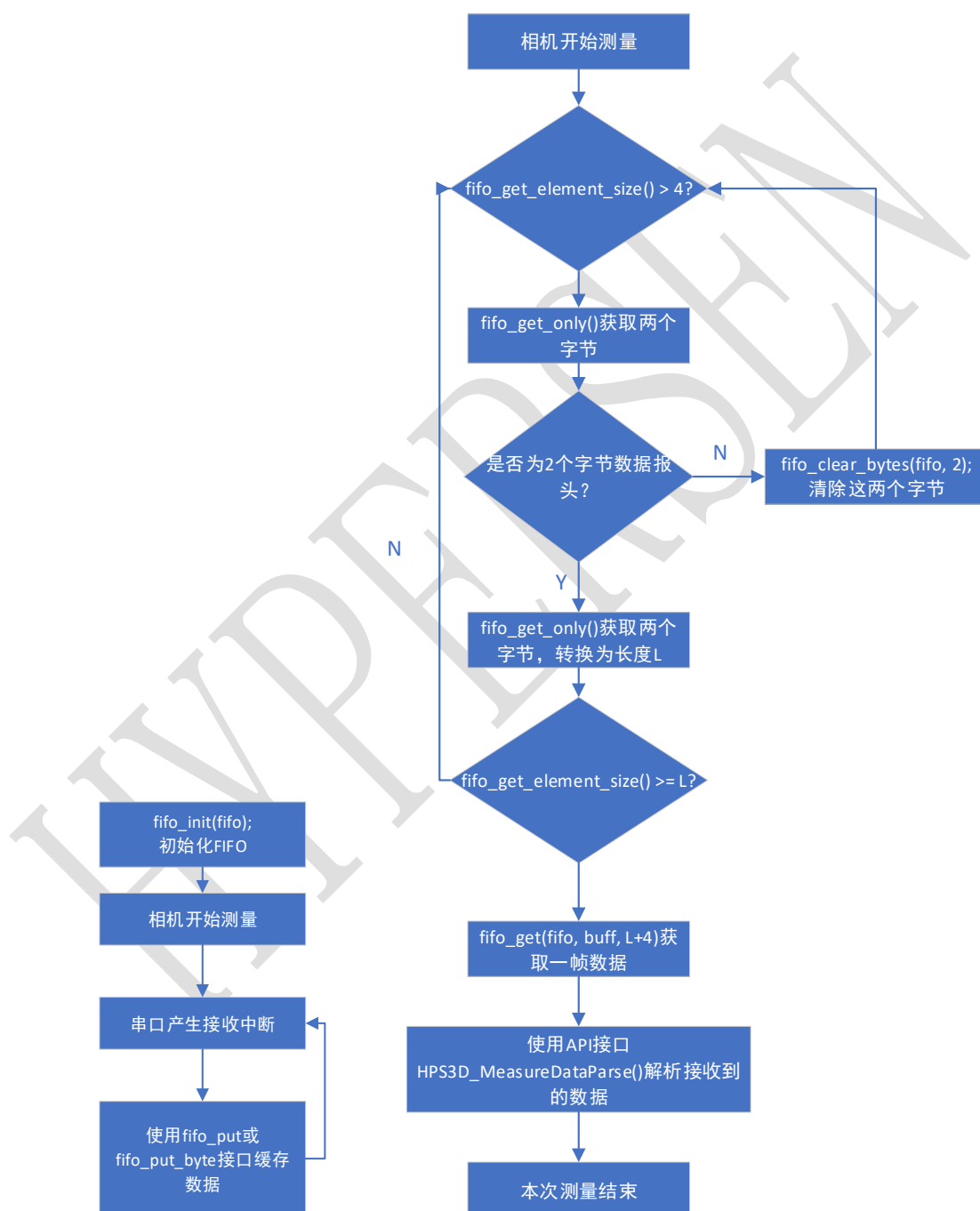
- 1) 中断+超时机制接收测量数据

这个接收机制的优势在于不需要额外的数据缓冲区，但是对于外设中断比较多的环境可能会出现丢帧现象，所以该机制比较适合负载低的环境或 RAM 吃紧的平台，具有较高的内存空间利用率，但牺牲了时间利用率。



## 2) 数据异步缓存机制

这个数据接收机制需要单独开辟一个数据缓冲区用于缓存数据，可使用循环 FIFO（SDK 测试程序已提供 fifo 模块），对于负载相对较高的环境可极大缓解来不及接收数据引起的丢帧现象，该机制的实现需要开启串口中断，串口每次产生接收中断就将数据缓存到 FIFO 中，主程序只需要对 FIFO 的数据长度进行检查判断，可充分利用方式 1) 的等待时间，具有更高的时间利用率，但牺牲了空间利用率。

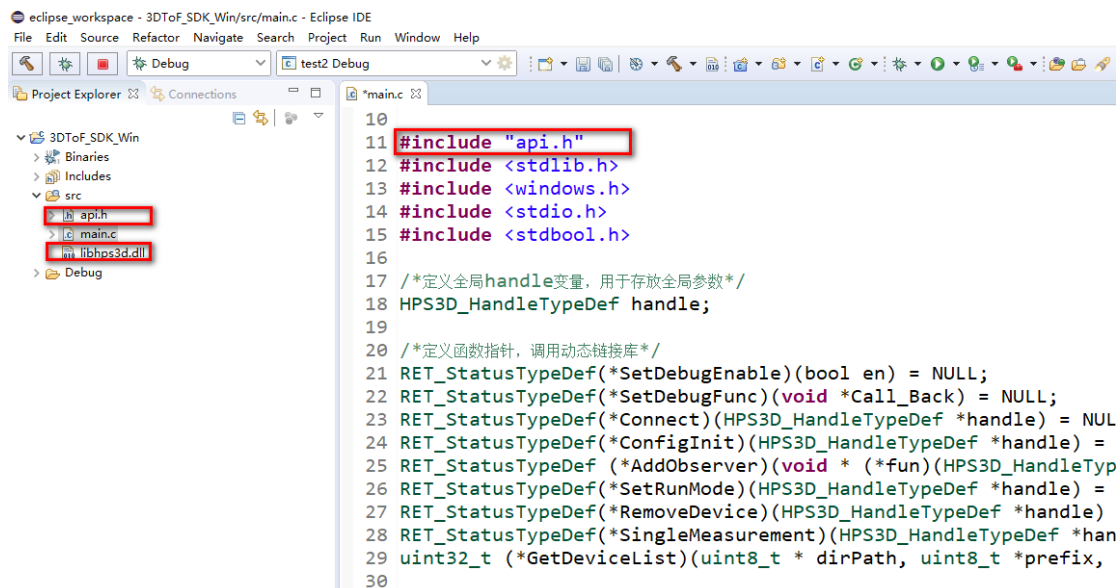


## 2.3 Windows 平台下将 SDK 集成到 IDE 中

xxx.dll 适合在 Windows 操作系统平台使用，这里以 windows 下的 eclipse 为例。本示

例基于 API 版本号为 2018.12.08 V1.0.0 的 SDK 进行编写。

### 2.3.1 添加 xxx.dll 动态链接库与 api.h 头文件到项目中



xxx.dll 动态链接库文件也可放置与其他路径下,调用时填写绝对路径即可;

### 2.3.2 在用户项目中使用 SDK

以下示例采用 windows 下的 LoadLibraryA 函数接口加载动态链接库:

- 1、在项目中包含 api.h 头文件,将 api.h 头文件包含在工程目录下

```
#include "api.h"
```

- 2、定义使用到的函数指针

```
/*定义函数指针, 调用动态链接库*/
RET_StatusTypeDef (*SetDebugEnable)(bool en) = NULL;
RET_StatusTypeDef (*SetDebugFunc)(void *Call_Back) = NULL;
RET_StatusTypeDef (*Connect)(HPS3D_HandleTypeDef *handle) = NULL;
RET_StatusTypeDef (*ConfigInit)(HPS3D_HandleTypeDef *handle) = NULL;
RET_StatusTypeDef (*AddObserver)(void * (*fun)(HPS3D_HandleTypeDef *,
AsyncIOObserver_t *), HPS3D_HandleTypeDef *handle, AsyncIOObserver_t
*Observer_t) = NULL;
RET_StatusTypeDef (*SetRunMode)(HPS3D_HandleTypeDef *handle) = NULL;
RET_StatusTypeDef (*RemoveDevice)(HPS3D_HandleTypeDef *handle) = NULL;
RET_StatusTypeDef (*SingleMeasurement)(HPS3D_HandleTypeDef *handle,
MeasureDataTypeDef *MeasureData, RetPacketTypeDef *PacketType) = NULL;
uint32_t (*GetDeviceList)(uint8_t * dirPath, uint8_t *prefix, uint8_t
fileName[DEV_NUM][DEV_NAME_SIZE]) = NULL;
```

- 3、加载动态库并获取相应函数地址

```
/*加载动态链接库*/
HMODULE module = LoadLibraryA("libhps3d.dll");
DWORD error_id = GetLastError();
if (module == NULL)
```



```

{
    system("error load");
    return 0;
}

SetDebugEnable = (RET_StatusTypeDef(*) (bool en))GetProcAddress(module,
"HPS3D_SetDebugEnable");
SetDebugFunc = (RET_StatusTypeDef(*) (void
*Call_Back))GetProcAddress(module, "HPS3D_SetDebugFunc");
Connect = (RET_StatusTypeDef(*) (HPS3D_HandleTypeDef
*handle))GetProcAddress(module, "HPS3D_Connect");
ConfigInit = (RET_StatusTypeDef(*) (HPS3D_HandleTypeDef
*handle))GetProcAddress(module, "HPS3D_ConfigInit");
AddObserver = (RET_StatusTypeDef(*) (void* (*fun) (HPS3D_HandleTypeDef *,
AsyncIOobserver_t *), HPS3D_HandleTypeDef *, AsyncIOobserver_t
*))GetProcAddress(module, "HPS3D_AddObserver");
SetRunMode = (RET_StatusTypeDef(*) (HPS3D_HandleTypeDef
*handle))GetProcAddress(module, "HPS3D_SetRunMode");
RemoveDevice = (RET_StatusTypeDef(*) (HPS3D_HandleTypeDef
*handle))GetProcAddress(module, "HPS3D_RemoveDevice");
SingleMeasurement = (RET_StatusTypeDef(*) (HPS3D_HandleTypeDef *,
MeasureDataTypeDef *, RetPacketTypeDef *))GetProcAddress(module,
"HPS3D_SingleMeasurement");
GetDeviceList = (uint32_t(*) (uint8_t *, uint8_t *, uint8_t
fileName[DEV_NUM][DEV_NAME_SIZE]))GetProcAddress(module,
"HPS3D_GetDeviceList");

```

4 调用相应的接口函数进行设备连接,设备初始化配置等操作,详细示例请参考 Windows 平台下的 Demo 程序;

#### ①设备连接

```

handle.DeviceName = "\\.\COMxx"; /*设置设备连接的端口号*/
ret = Connect(&handle); /*设备连接*/
if (ret != RET_OK)
{
    printf("设备连接失败! ret = %d\n", ret);
}

```

②设备初始化,默认配置为待机模式以及异步通信方式, 主要实现创建异步线程, 获取设备地址(其余命令均使用到设备地址),分配内存空间等

```

/*设备初始化*/
ret = ConfigInit(&handle);
if (RET_OK != ret)
{
    printf("初始化失败:%d\n", ret);
}
printf("初始化成功\n");

```

③设置测量模式: 单次测量或连续测量

```
/*连续测量*/  
handle.RunMode = RUN_CONTINUOUS;  
HPS3D_SetRunMode(&handle);  
  
/*单次测量*/  
Handle.RunMode = RUN_SINGLE_SHOT;  
HPS3D_SetRunMode(&handle);
```

按照以上步骤配置完成后即可得到默认配置下测量输出的完整深度数据包；

其中，连续模式需要异步线程来连续输出，此时可使用观察者模式来监听测量返回数据，示例代码如下：

```
/*用户观察者回调函数*/  
void* User_Fun(HPS3D_HandleTypeDef *handle, AsyncIOobserver_t *event)  
{  
    if(event->AsyncEvent == ISubject_Event_DataRecvd)  
    {  
        switch(event->RetPacketType)  
        {  
            case SIMPLE_ROI_PACKET:  
                printf("测量平均距离值  
为%d\n", event->MeasureData.simple_roi_data[0].distance_average);  
                break;  
            case FULL_ROI_PACKET:  
                printf("测量平均距离值  
为%d\n", event->MeasureData.full_roi_data[0].distance_average);  
                break;  
            case FULL_DEPTH_PACKET:  
                printf("测量平均距离值  
为%d\n", event->MeasureData.full_depth_data->distance_average);  
                break;  
            case SIMPLE_DEPTH_PACKET:  
                printf("测量平均距离值  
为%d\n", event->MeasureData.simple_depth_data->distance_average);  
                break;  
            case NULL_PACKET:  
                break;  
            default:  
                printf("系统错误\n");  
                break;  
        }  
    }  
}  
  
/*定义观察者并初始化*/  
AsyncIOobserver_t My_Observer;
```

```
/*观察者订阅事件为数据接收事件,即有数据则立即通知观察者*/
My_Observer.AsyncEvent = ISubject_Event_DataRecvd;
My_Observer.NotifyEnable = true; /*使能观察者*/
My_Observer.ObserverID = 2; /*观察者ID*/

/*添加观察者*/
ret = HPS3D_AddObserver(&User_Fun, &handle, &My_Observer);
if(RET_OK != ret)
{
    printf("观察者添加失败:%d\n", ret);
}
```

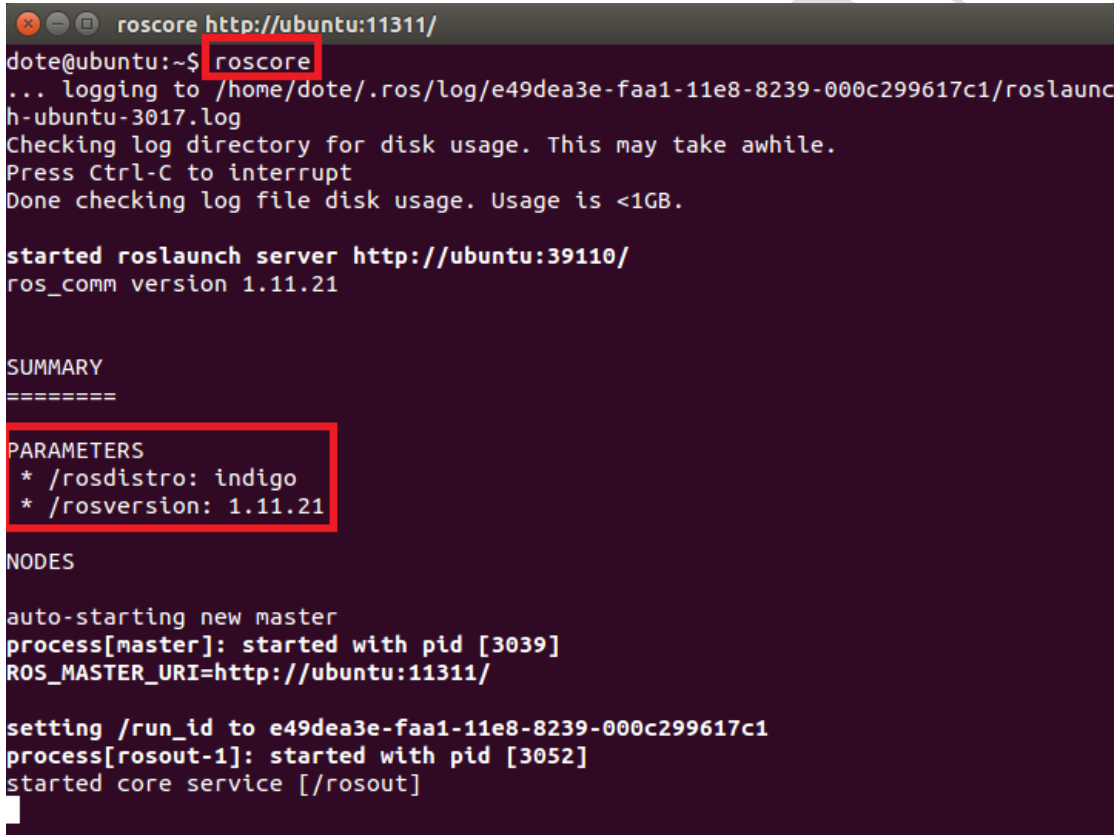
单次测量模式可支持同步也可支持异步方式,异步方式同上,建议使用同步方式;同步方式示例如下: **注: 同步单次测量函数会将 handle.SyncMode 设置为 SYNC 同步方式, 如需切换至异步方式需将该参数设置为 ASYNC;**

```
ret = HPS3D_SingleMeasurement(&handle);
if(ret == RET_OK)
{
    switch(handle.RetPacketType)
    {
        case SIMPLE_ROI_PACKET:
            printf("单次测量平均距离值为%d\n", handle.MeasureData.simple_roi_data[0].distance_average);
            break;
        case FULL_ROI_PACKET:
            printf("单次测量平均距离值为%d\n", handle.MeasureData.full_roi_data[0].distance_average);
            break;
        case FULL_DEPTH_PACKET:
            printf("单次测量平均距离值为%d\n", handle.MeasureData.full_depth_data->distance_average);
            break;
        case SIMPLE_DEPTH_PACKET:
            printf("单次测量平均距离值为%d\n", handle.MeasureData.simple_depth_data->distance_average);
            break;
        case NULL_PACKET:
            printf("无测量数据包\n");
            break;
        default:
            printf("系统错误\n");
            break;
    }
}
```

## 2.4 ROS 平台下环境配置及将 SDK 集成到 IDE 中

SDK ROS 提供了 HPS3D160 深度相机的应用程序接口，生成的 32 位/64 位的.so 动态链接库适合 Linux 操作系统上 ROS 平台使用，通过的.so 和 api.h 接口可集成进用户的项目源码进行编译。SDK 为二次开发包，仅提供基础的操作接口。可以获得深度图和完整 ROI（敏感区域）数据，若需要转换为点云数据，则在获取数据前，使能转换点云数据的接口即可。本文档基于 API 版本号为 2018.12.10 V1.0.0 的 SDK 进行编写。

SDK ROS 适合在 Linux 操作系统上 ROS 平台使用，这里以 Ubuntu14.04 为例。因为安装的是 Ubuntu14.04 的 Linux 操作系统，所以安装与之对应的 ROS 版本为发行版本 indigo，这里安装 1.11.21 版本。在终端输入 roscore，运行 ROS 总线，可查看运行 ROS 的版本，如下图所示：



```
roscore http://ubuntu:11311/
dote@ubuntu:~$ roscore
... logging to /home/dote/.ros/log/e49dea3e-faa1-11e8-8239-000c299617c1/roslaunch
h-ubuntu-3017.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:39110/
ros_comm version 1.11.21

SUMMARY
=====
PARAMETERS
* /rostdistro: indigo
* /rosversion: 1.11.21

NODES

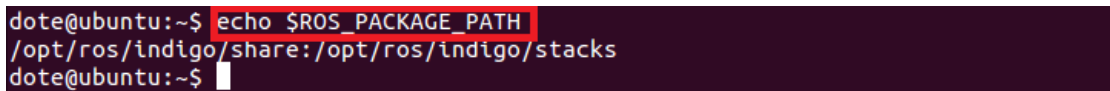
auto-starting new master
process[master]: started with pid [3039]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to e49dea3e-faa1-11e8-8239-000c299617c1
process[rosout-1]: started with pid [3052]
started core service [/rosout]
```

设备连接请参照本文档的“2.1.1 HPS3D160 设备连接”的内容。

### 2.4.1 创建一个工作空间

在创建一个工作空间前，首先查看环境变量，在终端输入 echo \$ROS\_PACKAGE\_PATH，查看 Linux 上的环境变量，如下图所示：



```
dote@ubuntu:~$ echo $ROS_PACKAGE_PATH
/opt/ros/indigo/share:/opt/ros/indigo/stacks
dote@ubuntu:~$
```

然后要检查是否安装 catkin 工具，若没有安装的话，请先安装 catkin 工具。默认情况下，是有 catkin 工具的，catkin 是 ROS 的一个官方的编译构建系统，是原本的 ROS 的编译构建系统 rosbuilt 的后继者。

(1) 生效 ROS 系统的环境变量，在终端输入 source /opt/ros/indigo/setup.bash。

- (3) 初始化工作空间，在 `src` 目录下执行 `catkin_init_workspace`，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ cd ~/HPS3D_SDK_ROS_Demo/src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ catkin_init_workspace
Creating symlink "/home/dote/HPS3D_SDK_ROS_Demo/src/CMakeLists.txt" pointing to
"/opt/ros/indigo/share/catkin/cmake/toplevel.cmake"
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$
```

- (4) 在工作空间根目录下输入 `catkin_make`，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ cd src/
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ cd ~/HPS3D_SDK_ROS_Demo/src
```

- (5) 此时工作空间下生成了 “build” 和 “devel” 两个文件夹。在 `devel` 文件夹下，您可以看到很多 `setup.*sh` 文件。输入 `source devel/setup.bash`，配置您的工作空间，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls
build devel src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls devel/
env.sh lib setup.bash setup.sh      setup_util.py setup.zsh
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ source devel/setup.bash
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$
```

**注：**任何的源文件、python 库、脚本，以及其他的静态文件，将会留在源空间 `src` 中。然而所有产生的文件，如库文件、可执行文件以及产生的代码都被放置在 `devel` 中。

## 2.4.2 创建一个 ROS 包 (catkin 包)

- (1) 创建一个名字为 `hps_camera` 的包, 它直接依赖于以下三个包: `std_msgs`, `rospy` 以及 `roscpp`, 进入 `src` 目录下, 再输入 `catkin create pkg hps_camera std_msgs rospy roscpp`, 如下图所示:

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ cd ~/HPS3D_SDK_ROS_Demo/src/  
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ catkin_create_pkg hps_camera std_msgs rosp  
y roscpp  
Created file hps_camera/CMakeLists.txt  
Created file hps_camera/package.xml  
Created folder hps_camera/include/hps_camera  
Created folder hps_camera/src  
Successfully created files in /home/dote/HPS3D_SDK_ROS_Demo/src/hps_camera. Plea  
se adjust the values in package.xml.  
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$
```

- (2) 在终端输入 `rospack depends hps_camera`，可以查看到 ROS 包的依赖关系，可看到本身加的依赖的三个包：`std_msgs`、`rospy` 和 `roscpp`，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ rosdep depends hps_camera  
cpp_common  
rostime  
roscpp_traits  
roscpp_serialization  
catkin  
genmsg  
genpy  
message_runtime  
gencpp  
genlisp  
message_generation  
roscpp  
std_msgs  
rosgraph_msgs  
xmlrpcpp  
roscpp  
rosgraph  
rospack  
roslib  
rospy  
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$
```

**注：**如果在执行 `rospack depends hps_camera` 时，出现如下错误（**在使用 ros\*命令时，出现错误，有可能就是工作空间失效了，重新生效即可**，也可将其写入 `ros` 环境变量中），则回到工作空间目录，再次执行 `source devel/setup.bash`，生效工作空间即可，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ rospack depends hps_camera
[rospack] Error: no such package hps_camera
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ cd ..
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls
build  devel  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ source devel/setup.bash
```

## 2.4.3 创建 ROS 消息 msg 和服务 srv

`msg` 文件是描述 ROS 中消息的域的简单的文本文件，它用来为消息产生不同语言的源代码。

`srv` 文件描述一个服务，它由两部分组成，请求和服务。

**提示：**一个 `msg` 文件或一个 `srv` 文件相当于一个结构体，所以可以对照提供的 `api.h`，在深度相机数据返回包内有五种数据类型，每一种数据类型都是一个结构体，即在写 `msg` 文件时，可以进行嵌套。

### 1、创建 ROS 消息 msg

（1）在创建的包中，创建一个消息 `msg` 目录，存放 `msg` 文件。创建 `msg` 目录：输入 `echo "uint16 distance_average"> msg/distance.msg`，创建 `distance.msg` 文件，并写入平均距离的变量。如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src$ cd ~/HPS3D_SDK_ROS_Demo/src/hps_camera/
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ ls
CMakeLists.txt  include  package.xml  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ mkdir msg
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ ls
CMakeLists.txt  include  msg  package.xml  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ echo "uint16 distance_average" > msg/distance.msg
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ sudo cat msg/distance.msg
uint16 distance_average
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$
```

（2）修改 `package.xml`，在文件中添加以下两句代码，如下图所示：

```
<build_depend> message_generation </build_depend>
<exec_depend> message_runtime </exec_depend>

<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
<build_depend> message_generation </build_depend>
<exec_depend> message_runtime </exec_depend>
```

到这里 `package.xml` 配置完成，保存并退出。

（3）配置 `CMakeLists.txt`，找到相应位置进行修改。

①添加 `message_generation` 到如下代码片，添加后结果如下图所示：



```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

②添加 CATKIN\_DEPENDS message\_runtime 到如下代码，添加后结果如下图所示：

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES hps_camera
#  CATKIN_DEPENDS roscpp rospy std_msgs
#  DEPENDS system lib
  CATKIN_DEPENDS message_runtime
)
```

③找到如下图代码片：

```
## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

将其添加刚创建的 distance.msg 消息文件，如下图所示：

```
## Generate messages in the 'msg' folder
add_message_files(
  FILES
  distance.msg
)
```

④找到如下图代码片：

```
## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

将其修改为如下图所示：

```
## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

到这里 CMakeLists.txt 配置完成，保存并退出。

(4) 查看刚刚创建的 msg 消息，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ rosmmsg show distance
[hps_camera/distance]:
uint16 distance_average
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$
```

## 2、创建 ROS 服务 srv



(1) 在创建的包中，创建一个服务 `srv` 目录，存放 `srv` 文件。创建 `camera.srv` 文件，并输入如下代码，如下图所示：

```
string client_node_name
---
string control_cmd
```

**注：**“string client\_node\_name”是请求，存储客户端节点的名称发送给服务器，“---”是分隔请求和响应，“string control\_cmd”是响应，存储服务器发送给客户端的控制命令。

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ cd ~/HPS3D_SDK_ROS_Demo/src/hps_camera/
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ ls
CMakeLists.txt  CMakeLists.txt~  include  msg  package.xml  package.xml~  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ mkdir srv
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ ls
CMakeLists.txt  CMakeLists.txt~  include  msg  package.xml  package.xml~  src  srv
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ sudo gedit srv/camera.srv
```

(2) 配置 `CMakeLists.txt`，找到相应位置进行修改。找到如下图代码段：

```
## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

将其添加刚刚创建的 `camera.srv` 服务文件，如下图所示：

```
## Generate services in the 'srv' folder
add_service_files(
  FILES
  camera.srv
)
```

到这里 `CMakeLists.txt` 配置完成，保存并退出。

(3) 查看刚刚创建的 `srv` 服务，在终端输入 `rossrv show camera`，如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera/srv$ rossrv show camera
[hps_camera/camera]:
string client_node_name
---
string control_cmd

dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera/srv$
```

### 3、重新构建 ROS 包

在工作目录中，输入 `catkin_make`，进行构建，将会看到如下图所示：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ cd ../../
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls
build  devel  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ catkin_make
```

```
#### Running command: "make -j1 -l1" in "/home/dote/HPS3D_SDK_ROS_Demo/build"
####
Scanning dependencies of target _hps_camera_generate_messages_check_deps_distance
[ 0%] Built target _hps_camera_generate_messages_check_deps_distance
Scanning dependencies of target _hps_camera_generate_messages_check_deps_camera
[ 0%] Built target _hps_camera_generate_messages_check_deps_camera
Scanning dependencies of target std_msgs_generate_messages_py
[ 0%] Built target std_msgs_generate_messages_py
Scanning dependencies of target hps_camera_generate_messages_py
[ 12%] Generating Python from MSG hps_camera/distance
[ 25%] Generating Python code from SRV hps_camera/camera
[ 37%] Generating Python msg __init__.py for hps_camera
[ 50%] Generating Python srv __init__.py for hps_camera
[ 50%] Built target hps_camera_generate_messages_py
Scanning dependencies of target std_msgs_generate_messages_lisp
[ 50%] Built target std_msgs_generate_messages_lisp
Scanning dependencies of target hps_camera_generate_messages_lisp
[ 62%] Generating Lisp code from hps_camera/distance.msg
[ 75%] Generating Lisp code from hps_camera/camera.srv
[ 75%] Built target hps_camera_generate_messages_lisp
Scanning dependencies of target std_msgs_generate_messages_cpp
[ 75%] Built target std_msgs_generate_messages_cpp
Scanning dependencies of target hps_camera_generate_messages_cpp
[ 87%] Generating C++ code from hps_camera/distance.msg
[100%] Generating C++ code from hps_camera/camera.srv
[100%] Built target hps_camera_generate_messages_cpp
Scanning dependencies of target hps_camera_generate_messages
[100%] Built target hps_camera_generate_messages
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$
```

#### 2.4.4 创建 ROS 的深度相机客户端节点和服务器

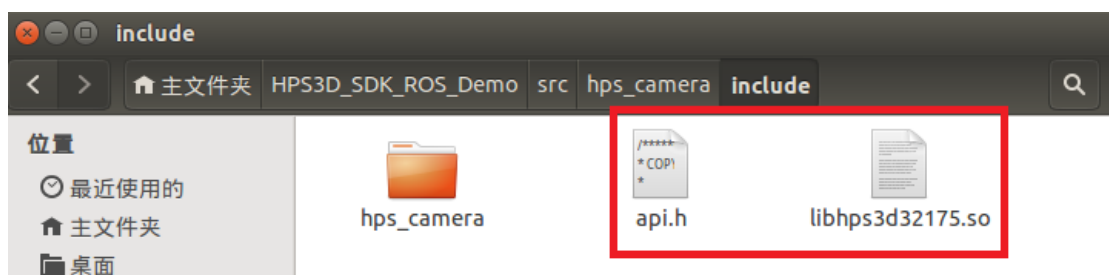
在这部分会提供例程，用户可根据需要自行修改或编写自己的程序，本文档给出的深度相机客户端节点和服务器的例程的流程是：

- (1) 客户端节点，配置好用户需采集的数据（或通过服务器发送的命令进行配置）；
- (2) 在客户端登陆后，连接可选设备文件，连接成功之后，向服务器发送自身的客户端名称（自定义名称）；
- (3) 在服务器接收到深度相机客户端发送的消息（客户端名称）之后，进行名称判断，是不是深度相机客户端，若是则向客户端发送开始命令（自定义命令），否则继续等待客户端连接发送的消息；
- (4) 在客户端接收到服务器发送过来的命令时，判断是什么命令，若是开始命令，则开使采集数据，并向服务器发布消息；
- (5) 服务器接收客户端节点发送的消息，可进行进一步匹配，设置等。

##### 1、将 api.h 和 .so 文件集成进工程

###### 方法 1:

- (1) 将 api.h 和 lib\*.so 复制到 include 目录下，如下图所示：



(2) 将 lib\*.so 复制到 /usr/local/lib 中, 输入 sudo ldconfig, 进行加载配置, 如下图所示:

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ ls
CMakeLists.txt  include  package.xml  src
CMakeLists.txt~ msg      package.xml~  srv
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ sudo mv include/libhps3d32175.so /usr/local/lib/
[sudo] password for dote:
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ sudo ldconfig
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$
```

(3) 修改 CMakeLists.txt 文件, 找到如下代码:

```
## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
  # include
    ${catkin_INCLUDE_DIRS}
)
```

将其修改为如下图所示:

```
## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
  include
    ${catkin_INCLUDE_DIRS}
)
```

到这里 CMakeLists.txt 配置完成, 保存并退出。:

方法 2:

在 ROS 包目录下创建 include 目录以及 lib 目录, 将 api.h 拷贝至 include 目录下, xxx.so 拷贝至 lib 目录下, 并修改 CmakeLists.txt 文件, 添加如下代码: 注意库名称应与 lib 目录下库名匹配;

```
#####
## Build ##
#####

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
  include
    ${catkin_INCLUDE_DIRS}
)
link_directories(
  lib
    ${catkin_LIB_DIRS}
)

## Declare a C++ library
# add_library(${PROJECT_NAME}
#   src/${PROJECT_NAME}/hps_camera.cpp
# )

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
## either from message generation or dynamic reconfigure
# add_dependencies(${PROJECT_NAME} ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

find_package(PCL REQUIRED)
include_directories(include${PCL_INCLUDE_DIRS})
add_executable(ros_camera_client src/ros_camera_client.cpp)
target_link_libraries(ros_camera_client ${catkin_LIBRARIES} ${PCL_LIBRARIES} hps3d)
```

修改完成即可;

## 2、创建 ROS 的深度相机客户端节点和服务器

(1) 在 src 目录中, 创建 ros\_camera\_client.cpp 和 ros\_camera\_server.cpp, 如下图所示:

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ ls
CMakeLists.txt  include  package.xml  src
CMakeLists.txt~ msg      package.xml~ srv
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ sudo touch src/ros_camera_client.cpp src/ros_camera_server.cpp
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$ sudo ls src/
ros_camera_client.cpp  ros_camera_server.cpp
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/src/hps_camera$
```

(4) 修改 CMakeLists.txt 文件，添加如下代码，如下图所示：

```
add_executable(ros_camera_client src/ros_camera_client.cpp)
target_link_libraries(ros_camera_client ${catkinhps_camera_LIBRARIES} hps3d32175)
add_dependencies(ros_camera_client hps_camera_generate_messages_cpp)

add_executable(ros_camera_server src/ros_camera_server.cpp)
target_link_libraries(ros_camera_server ${catkin_LIBRARIES})
add_dependencies(ros_camera_server hps_camera_generate_messages_cpp)
```

```
## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
```

```
## Declare a C++ library
# add_library(${PROJECT_NAME}
#   src/${PROJECT_NAME}/hps_camera.cpp
# )
```

```
add_executable(ros_camera_client src/ros_camera_client.cpp)
target_link_libraries(ros_camera_client ${catkin_LIBRARIES} hps3d32175)
add_dependencies(ros_camera_client hps_camera_generate_messages_cpp)

add_executable(ros_camera_server src/ros_camera_server.cpp)
target_link_libraries(ros_camera_server ${catkin_LIBRARIES})
add_dependencies(ros_camera_server hps_camera_generate_messages_cpp)
```

### 3、编写 ROS 的深度相机客户端节点和服务端

(1) 编写服务器程序

①添加头文件，代码如下：

```
#include "ros/ros.h"//ros
#include "hps_camera/distance.h"//msg
#include "hps_camera/camera.h"//srv
```

②在主函数中进行 ros 初始化，节点创建，话题创建等，代码如下：

```
ros::init(argc, argv, "ros_camera_server");
ros::NodeHandle n;
ros::ServiceServer service = n.advertiseService("client_login", send_cmd);
ros::Subscriber sub = n.subscribe("camera", 1000, chatterCallback);
ros::spin();
```

③服务函数的编写，代码如下：

```
bool send_cmd(hps_camera::camera::Request &req, hps_camera::camera::Response &res)
```

```
{
    std::stringstream scmd;
    printf("client_name: %s\n", req.client_node_name.c_str() );
    if( strcmp(req.client_node_name.c_str(), "camera_client" ) == 0 )
    {
        scmd<< "start";
        res.control_cmd = scmd.str();
        printf("send_cmd: %s\n", res.control_cmd.c_str() );
    }
    return true;
}
```

④订阅话题的回调函数，代码如下：

```
void chatterCallback(const hps_camera::distance& msg)
{
    printf("distance_average = %d\n", msg.distance_average);
}
```

(2) 编写深度相机客户端节点程序

在客户端节点中，深度相机 api 接口的使用与本文档的“2.1.3 在用户项目中使用 SDK”配置一样，详情请参照本文档的“2.1.3 在用户项目中使用 SDK”的内容或给出的示例代码。

①添加头文件，代码如下：

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "hps_camera/distance.h"//msg
#include "hps_camera/camera.h"//srv
#include "../include/api.h"
#include <sstream>
```

②在主函数中，ros 初始化，节点创建，话题创建以及深度相机 api 接口配置，代码如下：

```
ros::Publisher camera_pub;//全局变量，因为在观察者回调函数需要使用
int main(int argc, char **argv)
{
    ros::init(argc, argv, "ros_camera_client");
    ros::NodeHandle n;
    .....
    std::stringstream sclient_name;
    ros::ServiceClient client =
n.serviceClient<hps_camera::camera>("client_login");
    hps_camera::camera srv;
    sclient_name<<"camera_client";
    printf("send name = %s\n",sclient_name.str().c_str());
    srv.request.client_node_name = sclient_name.str();
    camera_pub = n.advertise<hps_camera::distance>("camera", 1000);
    .....
    if (client.call(srv))
    {
```

```

while(ros::ok())
{
    printf("rev cmd = %s\n", srv.response.control_cmd.c_str());
    if( strcmp(srv.response.control_cmd.c_str(), "start" ) == 0 )
    {
        break;
    }
}

else
{
    break;
}

.....
while(1)
{
    ros::spinOnce();
}

return 0;
}

```

③在本文档的“2.1.3 在用户项目中使用 SDK”中，第 5 点（2）异步测量的②编写观察者回调函数，需要在测量得到数值时，将数值赋值给 msg 消息结构体，并发布。代码如下：

```

/*观察者回调函数*/
void *User_Func(HPS3D_HandleTypeDef *handle, AsyncIOobserver_t *event)
{
    hps_camera::distance msg;
    if(event->AsyncEvent == ISubject_Event_DataRecvd)
    {
        switch(event->RetPacketType)
        {
            case SIMPLE_ROI_PACKET:
                printf("distance = %d  event->RetPacketType = %d\n",
event->MeasureData.simple_roi_data[0].distance_average, event->RetPacketType);
                break;
            case FULL_ROI_PACKET:
                msg.distance_average =
event->MeasureData.full_roi_data[0].distance_average;
                printf("distance = %d\n", msg.distance_average);
                camera_pub.publish(msg);
                break;
            case FULL_DEPTH_PACKET:
                printf("distance = %d  event->RetPacketType = %d\n",
event->MeasureData.full_depth_data->distance_average, event->RetPacketType);
                break;
        }
    }
}

```



```

    case SIMPLE_DEPTH_PACKET:
        printf("distance = %d  event->RetPacketType = %d\n",
event->MeasureData.simple_depth_data->distance_average, event->RetPacketType);
        break;
    case NULL_PACKET:
        /*返回数据包类型为空*/
        break;
    default:
        printf("system error!\n");
        break;
}
event->RetPacketType = NULL_PACKET;
}
return 0;
}

```

#### 4、测试 ROS 的深度相机客户端节点和服务端

(1) 编译工程，在/devel/lib/hps\_camera 文件夹下可以看到执行文件，如下图所示：

```

dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls
build  devel  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ cd devel/lib/hps_camera/
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/devel/lib/hps_camera$ ls
ros_camera_client  ros_camera_server
dote@ubuntu:~/HPS3D_SDK_ROS_Demo/devel/lib/hps_camera$

```

(2) 修改 dev/ttyACM\*设备文件权限，如下图所示：

```

dote@ubuntu:~$ cd /dev/
dote@ubuntu:/dev$ ll ttyACM*
-rw-rw---- 1 root dialout 166, 0 12月 10 20:40 ttyACM0
dote@ubuntu:/dev$ sudo chmod 777 ttyACM0
[sudo] password for dote:
dote@ubuntu:/dev$ ll ttyACM*
crwxrwxrwx 1 root dialout 166, 0 12月 10 20:40 ttyACM0
dote@ubuntu:/dev$

```

(3) 新打开一个终端输入 roscore，运行 ros 总线，如下图所示：

```

dote@ubuntu:~$ roscore
... logging to /home/dote/.ros/log/5f389700-fcf3-11e8-9823-000c299617c1/roslaunch
h-ubuntu-14026.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:37024/
ros_comm version 1.11.21

```

(4) 新打开两个终端，进入到工作空间，输入 source devel/setup.bash，生效工作空间。

①在终端运行服务器，如下图所示：

```

dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls
build  devel  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ source devel/setup.bash
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ roslaunch hps_camera ros_camera_server
waiting client login

```

②在另一个终端运行客户端节点，如下图所示：



```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ ls
build  devel  src
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ source devel/setup.bash
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ rosrn hps_camera ros_camera_client
send name = camera_client
当前可连接的设备(请选择):
0: /dev/ttyACM0
请输入对应的序号:
```

③在客户端终端选择可连接设备，输入 0，得到如下结果：

客户端终端：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ rosrn hps_camera ros_camera_client
send name = camera_client
当前可连接的设备(请选择):
0: /dev/ttyACM0
请输入对应的序号:
0
初始化成功
rev cmd = start
distance = 1806
distance = 1797
distance = 1817
distance = 1798
distance = 1802
distance = 1793
```

服务器终端：

```
dote@ubuntu:~/HPS3D_SDK_ROS_Demo$ rosrn hps_camera ros_camera_server
waiting client login
client_name: camera_client
send_cmd: start
distance_average = 1806
distance_average = 1797
distance_average = 1817
distance_average = 1798
distance_average = 1802
distance_average = 1793
```

## 三、API 函数接口

### 3.1 设置运行模式

运行模式主要包含待机模式(停止测量)、单次测量模式以及连续测量模式；

#### 3.1.1 示例代码

```
handle.RunMode = RUN_CONTINUOUS;
ret = HPS3D_SetRunMode(&handle);
```

### 3.2 获取/设置设备地址

设备地址是多机协作时用于区分设备的编号，同时也是帧 ID，此参数可修改；

#### 3.2.1 示例代码

```
ret = HPS3D_GetDevAddr(&handle);
printf("1handle.DeviceAddr = %#x\n", handle.DeviceAddr);
ret = HPS3D_SetDevAddr(&handle, 0x01);
```

```
ret = HPS3D_GetDevAddr(&handle);  
printf("2handle.DeviceAddr = %#x\n", handle.DeviceAddr);
```

### 3.2.2 运行结果

```
1handle.DeviceAddr = 0  
2handle.DeviceAddr = 0x1
```

## 3.3 获取设备版本信息

设备版本信息分为固件版本信息以及 SDK 版本信息；注：仅 SDK 版本与设备版本匹配时才能正常使用，否则需要升级固件或更新 SDK 版本；

### 3.3.1 示例代码

```
Version_t version_t;  
HPS3D_GetDeviceVersion(&handle, &version_t);  
  
version_t = HPS3D_GetDeviceVersion();
```

### 3.3.2 运行结果

```
version_t.year = 18  
version_t.month = 11  
version_t.day = 15  
version_t.major = 1  
version_t.minor = 7  
version_t.rev = 9
```

## 3.4 获取/设置数据包类型

此处数据包设置主要分为简单包和完整包；其中简单包是指不包含深度信息的输出数据包，而完整包则包含所有像素点的深度信息参数；

### 3.4.1 示例代码

```
/*获取数据包类型*/  
HPS3D_GetPacketType(&handle);  
/*设置数据包类型*/  
handle.PacketType = PACKET_SIMPLE; /*设置成简单数据包*/  
HPS3D_SetPacketType(&handle);  
HPS3D_GetPacketType(&handle);  
  
/*设置测量数据包类型，需要在初始化完成前进行该配置，默认为完整深度数据包*/  
/*设置测量数据包类型*/  
typedef enum  
{  
    DEPTH_DATA_PACKET = 0x0, /*深度数据包*/  
    ROI_DATA_PACKET, /*ROI数据包*/  
    OBSTACLE_PACKET /*障碍物数据包*/  
}MeasurePacketTypeDef;  
HPS3D_SetMeasurePacketType(DEPTH_DATA_PACKET); /*设置为完整数据包*/  
/*获取测量数据包类型*/  
MeasurePacketTypeDef HPS3D_GetMeasurePacketType(void)
```

### 3.4.2 运行结果

```
1handle.PacketType = 0  
2handle.PacketType = 1
```

### 3.5 保存/清除用户配置、恢复出厂设置

保存用户设置是使得当前配置永久生效，调用此函数将会将当前配置写入到传感器的 Flash 中，重新上电后将以该配置作为默认配置；

#### 3.5.1 示例代码

```
HPS3D_ProfileSaveToCurrent(&handle); /*保存到用户配置，重新上电默认为此配置*/  
HPS3D_ProfileClearCurrent(&handle); /*清除用户配置，恢复到默认配置*/  
HPS3D_ProfileRestoreFactory(&handle); /*恢复出厂配置*/
```

### 3.6 获取传输类型

#### 3.6.1 示例代码

```
/*获取传输类型*/  
TransportTypeDef transport_type;  
HPS3D_GetTransportType(&handle, &transport_type);  
printf("transport_type = %d\n", transport_type); //TRANSPORT_USB = 0
```

#### 3.6.2 运行结果

```
transport_type = 0
```

### 3.7 选择 ROI 组/获取当前 ROI 组 ID

ROI 分组数量和 ROI 数量根据固件来确定，使用此函数可以切换到不同的分组下执行不同的参数配置；

#### 3.7.1 示例代码

```
uint8_t group_id = 0;  
/*选择组 ID 为 3*/  
HPS3D_SelectROIGroup(&handle, 3);  
  
/*获得组 ID*/  
HPS3D_GetROIGroupID(&handle, &group_id);  
printf("group_id = %d\n", group_id);
```

#### 3.7.2 运行结果

```
group_id = 3
```

### 3.8 ROI 相关配置

ROI 详细说明请参考固态面阵激光雷达客户端软件操作手册 (HPS3D\_RM001) 中的敏感区域设定章节，建议使用客户端软件将敏感区域设定好后保存到用户配置中，使其永久生效；不建议使用以下配置接口进行设定，如有必要请详细阅读一下配置示例代码；

#### 3.8.1 示例代码

```
ROIConfTypeDef roi_conf1, roi_conf2;
```

```
HysteresisSingleConfTypeDef hysteresis_conf1, hysteresis_conf2;

/*选择组 0, roi_id = 0, threshold_id = 2, GPIO 警报, ROI 区域的距离最小值作为参考值*/
/*选择组 0, roi_id = 2, threshold_id = 1, GPIO 警报关闭, ROI 区域的有效幅值平均值作为参考值*/
/*设置 ROI 警报类型*/
HPS3D_SetROIAlarmType(&handle, 0, 2, ROI_ALARM_GPIO);
HPS3D_SetROIAlarmType(&handle, 2, 1, ROI_ALARM_DISABLE);

/*设置 ROI 的参考值类型*/
HPS3D_SetROIReferenceType(&handle, 0, 2, ROI_REF_DIST_MIN);
HPS3D_SetROIReferenceType(&handle, 2, 1, ROI_REF_VALID_AMPLITUDE);

/*设定 ROI 区域*/
roi_conf1.roi_id = 0;
roi_conf1.left_top_x = 10;
roi_conf1.left_top_y = 10;
roi_conf1.right_bottom_x = 30;
roi_conf1.right_bottom_y = 20;
HPS3D_SetROIRegion(&handle, roi_conf1);

roi_conf2.roi_id = 2;
roi_conf2.left_top_x = 40;
roi_conf2.left_top_y = 30;
roi_conf2.right_bottom_x = 80;
roi_conf2.right_bottom_y = 50;
HPS3D_SetROIRegion(&handle, roi_conf2);

/*设置 ROI 使能*/
HPS3D_SetROIEnable(&handle, 0, true);
HPS3D_SetROIEnable(&handle, 2, true);

/*设置 ROI 阈值使能*/
HPS3D_SetROIThresholdEnable(&handle, 0, 2, true);
HPS3D_SetROIThresholdEnable(&handle, 2, 1, true);

/*设置 ROI 阈值配置*/
hysteresis_conf1.threshold_value = 20;
hysteresis_conf1.hysteresis = 100;
hysteresis_conf1.positive = true;
HPS3D_SetROIThresholdConf(&handle, 0, 2, 60, hysteresis_conf1);

hysteresis_conf2.threshold_value = 30;
```

```
hysteresis_conf2.hysteresis = 200;
hysteresis_conf2.positive = false;
HPS3D_SetROIThresholdConf(&handle, 2, 1, 70, hysteresis_conf2);

/*获取指定的 ROI 配置*/
HPS3D_GetROIConfById(&handle, 0, &roi_conf1);
HPS3D_GetROIConfById(&handle, 2, &roi_conf2);
```

### 3.8.2 运行结果

```
roi_conf1.roi_id = 0
roi_conf1.enable = 1
roi_conf1.left_top_x = 10
roi_conf1.left_top_y = 10
roi_conf1.right_bottom_x = 30
roi_conf1.right_bottom_y = 20
roi_conf1.alarm_type[2] = 1
roi_conf1.roi_conf1.ref_type[2] = 2
roi_conf1.roi_conf1.pixel_number_threshold[2] = 60
roi_conf1.hysteresis_conf[2].enable = 1
roi_conf1.hysteresis_conf[2].threshold_value = 20
roi_conf1.hysteresis_conf[2].positive = 1
roi_conf1.hysteresis_conf[2].hysteresis = 100

roi_conf2.roi_id = 2
roi_conf2.enable = 1
roi_conf2.left_top_x = 40
roi_conf2.left_top_y = 30
roi_conf2.right_bottom_x = 80
roi_conf2.right_bottom_y = 50
roi_conf2.alarm_type[1] = 0
roi_conf2.roi_conf1.ref_type[1] = 6
roi_conf2.hysteresis_conf[1].threshold_id = 2
roi_conf2.hysteresis_conf[1].enable = 1
roi_conf2.hysteresis_conf[1].threshold_value = 30
roi_conf2.hysteresis_conf[1].positive = 0
roi_conf2.hysteresis_conf[1].hysteresis = 200
```

## 3.9 获取当前设备支持的 ROI 数量和阈值数量

当前设备的 ROI 数量以及阈值数量是根据固件来决定的，在设备初始化时将会默认得到这两个参数；

### 3.9.1 示例代码

```
/*获取当前设备支持的 ROI 数量和阈值数量*/
uint8_t roi_number, threshold_number;
HPS3D_GetNumberOfROI(&handle, &roi_number, &threshold_number);
```

### 3.9.2 运行结果

```
roi_number = 20
threshold number = 3
```

## 3.10 设置/获取输出/输入配置

配置 GPIO 输入输出引脚相关功能，不建议在此进行设置，请参考固态面阵激光雷达客户端软件操作手册 (HPS3D\_RM001) 通过客户端软件进行操作；

### 3.10.1 示例代码

```
/*获取 GPOUT 的配置*/
GPIOOutConfTypeDef gpio_out_conf;
gpio_out_conf.gpio = GPOUT_1;
HPS3D_GetGPIOOutConf(&handle, &gpio_out_conf);

/*获取 GPIN 的配置*/
GPIOInConfTypeDef gpio_in_conf;
gpio_in_conf.gpio = GPIN_1;
HPS3D_GetGPIOInConf(&handle, &gpio_in_conf);

/*设置 GPOUT 的配置*/
gpio_out_conf.gpio = GPOUT_1;
gpio_out_conf.function = 1;
gpio_out_conf.polarity = 1;
HPS3D_SetGPIOOut(&handle, gpio_out_conf);
HPS3D_GetGPIOOutConf(&handle, &gpio_out_conf);

/*设置 GPIN 的配置*/
gpio_in_conf.gpio = GPIN_1;
gpio_in_conf.function = 0; /*注：这里功能如果设置为 1, 则命令接口失效，只能触发 IO 才生效!!! */
gpio_in_conf.polarity = 1;
HPS3D_SetGPIOIn(&handle, gpio_in_conf);
HPS3D_GetGPIOInConf(&handle, &gpio_in_conf);
```

### 3.10.2 运行结果

```
1gpio_out_conf.function = 0
1gpio_out_conf.polarity = 0
1gpio_in_conf.function = 0
1gpio_in_conf.polarity = 0
2gpio_out_conf.function = 1
2gpio_out_conf.polarity = 1
2gpio_in_conf.function = 0
2gpio_in_conf.polarity = 1
```

## 3.11 设置 HDR 模式

HDR 模式详细描述请参考固态面阵激光雷达客户端软件操作手册 (HPS3D\_RM001) 中 4.3 章节

### 3.11.1 示例代码

```
/*获得 HDR 模式*/
HDRConf hdr_conf;
HPS3D_GetHDRConfig(&handle, &hdr_conf);

/*设置成 AUTO_HDR*/
HPS3D_SetHDRMode(&handle, AUTO_HDR); /*AUTO_HDR = 1*/
```

```
HPS3D_GetHDRConfig(&handle, &hdr_conf);
```

### 3.11.2 运行结果

```
1hdr_conf.hdr_mode = 3  
2hdr_conf.hdr_mode = 1
```

## 3.12 设置/获取 HDR 配置

### 3.12.1 示例代码

```
/*获取距离滤波器配置*/  
HDRConf hdr_conf, set_conf;  
HPS3D_GetHDRConfig(&handle, &hdr_conf);  
  
/*1、设置 HDR, HDR-DISABLE*/  
set_conf.hdr_mode = HDR_DISABLE;  
set_conf.hdr_disable_integration_time = 1000;  
HPS3D_SetHDRConfig(&handle, set_conf);  
HPS3D_GetHDRConfig(&handle, &hdr_conf);  
  
/*2、设置 HDR, AUTO-HDR*/  
set_conf.hdr_mode = AUTO_HDR;  
set_conf.quality_overexposed = 600; /*曝光幅值一定要小于过度曝光赋值, 否则设置失败*/  
set_conf.quality_overexposed_serious = 900;  
set_conf.quality_weak = 80; /*弱幅值一定要大于极弱幅值, 否则设置失败*/  
set_conf.quality_weak_serious = 60;  
HPS3D_SetHDRConfig(&handle, set_conf);  
HPS3D_GetHDRConfig(&handle, &hdr_conf);  
  
/*3、设置 HDR, SIMPLE-HDR*/  
set_conf.hdr_mode = SIMPLE_HDR;  
set_conf.simple_hdr_max_integration = 500; /*必须大于最小积分时间, 否则设置失败*/  
set_conf.simple_hdr_min_integration = 400;  
HPS3D_SetHDRConfig(&handle, set_conf);  
HPS3D_GetHDRConfig(&handle, &hdr_conf);  
  
/*4、设置 HDR, SUPER-HDR*/  
set_conf.hdr_mode = SUPER_HDR;  
set_conf.super_hdr_frame_number = 2;  
set_conf.super_hdr_max_integration = 15000;  
HPS3D_SetHDRConfig(&handle, set_conf);  
HPS3D_GetHDRConfig(&handle, &hdr_conf);
```

### 3.12.2 运行结果



```
1hdr_conf.hdr_mode = 1
1hdr_conf.hdr_disable_integration_time = 400
1hdr_conf.quality_overexposed = 500.000000
1hdr_conf.quality_overexposed_serious = 800.000000
1hdr_conf.quality_weak = 90.000000
1hdr_conf.quality_weak_serious = 50.000000
1hdr_conf.simple_hdr_max_integration = 2000
1hdr_conf.simple_hdr_min_integration = 100
1hdr_conf.super_hdr_frame_number = 4
1hdr_conf.super_hdr_max_integration = 30000

2hdr_conf.hdr_mode = 0
2hdr_conf.hdr_disable_integration_time = 1000
3hdr_conf.hdr_mode = 1
3hdr_conf.quality_overexposed = 600.000000
3hdr_conf.quality_overexposed_serious = 900.000000
3hdr_conf.quality_weak = 80.000000
3hdr_conf.quality_weak_serious = 60.000000
4hdr_conf.hdr_mode = 3
4hdr_conf.simple_hdr_max_integration = 500
4hdr_conf.simple_hdr_min_integration = 400
5hdr_conf.hdr_mode = 2
5hdr_conf.super_hdr_frame_number = 2
5hdr_conf.super_hdr_max_integration = 15000
```

### 3.13 设置/获取距离滤波器配置

距离滤波器是提高重复精度的一种滤波算法，使用该配置将使得测量重复精度大幅度提升。默认参数可参考客户端软件 HPS3D-Client 中高级设定中距离滤波器参数配置；

#### 3.13.1 示例代码

```
/*获取距离滤波器配置*/
DistanceFilterConfTypeDef distance_filter_conf, set_conf;
HPS3D_GetDistanceFilterConf(&handle, &distance_filter_conf);

/*设置距离滤波器类型*/
HPS3D_SetDistanceFilterType(&handle, DISTANCE_FILTER_SIMPLE_KALMAN); /*简单的卡尔曼滤波器*/
HPS3D_GetDistanceFilterConf(&handle, &distance_filter_conf);

/*配置距离滤波器*/
set_conf.kalman_K = 0.3;
set_conf.kalman_threshold = 200;
set_conf.num_check = 3;
HPS3D_SetSimpleKalman(&handle, set_conf);
HPS3D_GetDistanceFilterConf(&handle, &distance_filter_conf);
```

#### 3.13.2 运行结果

```
1distance_filter_conf.filter_type = 0
1distance_filter_conf.kalman_K = 0.100000
1distance_filter_conf.kalman_threshold = 100
1distance_filter_conf.num_check = 2
2distance_filter_conf.filter_type = 1
2distance_filter_conf.kalman_K = 0.300000
2distance_filter_conf.kalman_threshold = 200
2distance_filter_conf.num_check = 3
```

## 3.14 设置/获取平滑滤波器配置

### 3.14.1 示例代码

```
/*获取平滑滤波器的配置*/  
SmoothFilterConfTypeDef smooth_filter_conf, set_conf;  
HPS3D_GetSmoothFilterConf(&handle, &smooth_filter_conf);  
  
/*设置平滑滤波器*/  
set_conf.type = SMOOTH_FILTER_AVERAGE; /*均值滤波器*/  
set_conf.arg1 = 200;  
HPS3D_SetSmoothFilter(&handle, set_conf);  
HPS3D_GetSmoothFilterConf(&handle, &smooth_filter_conf);
```

### 3.14.2 运行结果

```
1smooth_filter_conf.type = 0  
1smooth_filter_conf.arg1 = 0  
2smooth_filter_conf.type = 1  
2smooth_filter_conf.arg1 = 200
```

## 3.15 设置/获取光学参数

光学参数主要包含光学照明视场角以及可视角，光学照明视场角为内部使用，可视角是指当前设备可视角度，水平方向 76°，垂直方向 32°；

### 3.15.1 示例代码

```
OpticalParamConfTypeDef optical_param_conf;  
HPS3D_GetOpticalParamConf(&handle, &optical_param_conf);  
  
HPS3D_SetOpticalEnable(&handle, false);  
HPS3D_GetOpticalParamConf(&handle, &optical_param_conf);
```

### 3.15.2 运行结果

```
1optical_param_conf.enable = 1  
1optical_param_conf.illum_angle_horiz = 82  
1optical_param_conf.illum_angle_vertical = 36  
1optical_param_conf.viewing_angle_horiz = 76  
1optical_param_conf.viewing_angle_vertical = 32  
2optical_param_conf.enable = 0
```

## 3.16 设置/获取距离补偿

距离补偿是将当前实际距离与测量返回值之间的误差补偿，从而提高测量的准确性；例如当前实际距离为 1500mm，测量返回值为 1800mm；此时相差 300mm，则需将 offset 设置为-300 进行补偿；

### 3.16.1 示例代码

```
int16_t offset;  
HPS3D_GetDistanceOffset(&handle, &offset);  
  
HPS3D_SetDistanceOffset(&handle, 20);
```

```
HPS3D_GetDistanceOffset(&handle, &offset);
```

### 3.16.2 运行结果

```
1offset = 0  
2offset = 20
```

## 3.17 设置/获取畸变检测参数

畸变检测功能是为了检测异常测量数据,如出现异常测量结果该像素点将会被幅值为无效测量点;畸变检测参数默认即可,无需修改;如需使用该功能仅需调用 HPS3D\_SetInterferenceDetectEn(...)进行设能设置;

### 3.17.1 示例代码

```
InterferenceDetectConfTypeDef interference_detect_conf;  
HPS3D_GetInterferenceDetectConf(&handle, &interference_detect_conf);  
  
HPS3D_SetInterferenceDetectEn(&handle, true);  
HPS3D_GetInterferenceDetectConf(&handle, &interference_detect_conf);  
  
HPS3D_SetInterferenceDetectIntegTime(&handle, 1000);  
HPS3D_GetInterferenceDetectConf(&handle, &interference_detect_conf);  
  
HPS3D_SetInterferenceDetectAmplitudeThreshold(&handle, 100);  
HPS3D_GetInterferenceDetectConf(&handle, &interference_detect_conf);  
  
HPS3D_SetInterferenceDetectCaptureNumber(&handle, 2);  
HPS3D_GetInterferenceDetectConf(&handle, &interference_detect_conf);  
  
HPS3D_SetInterferenceDetectNumberCheck(&handle, 2);  
HPS3D_GetInterferenceDetectConf(&handle, &interference_detect_conf);
```

### 3.17.2 运行结果

```
1interference_detect_conf.enable = 0  
1interference_detect_conf.integ_time = 250  
1interference_detect_conf.amplitude_threshold = 6  
1interference_detect_conf.capture_num = 2  
1interference_detect_conf.number_check = 1  
2interference_detect_conf.enable = 1  
2interference_detect_conf.integ_time = 200  
2interference_detect_conf.amplitude_threshold = 5  
2interference_detect_conf.capture_num = 6  
2interference_detect_conf.number_check = 3
```

## 3.18 设置/获取安装角度参数

此功能暂时无效,无需配置;

### 3.18.1 示例代码

```
MountingAngleParamTypeDef mounting_angle_param_conf, set_conf;  
HPS3D_GetMountingParamConf(&handle, &mounting_angle_param_conf);  
  
HPS3D_SetMountingAngleEnable(&handle, true);
```

```
set_conf.angle_vertical = 50;/*50 度*/  
HPS3D_SetMountingAngleParamConf(&handle, set_conf);  
  
HPS3D_GetMountingParamConf(&handle, &mounting_angle_param_conf);
```

### 3.18.2 运行结果

```
1mounting_angle_param_conf.enable = 0  
1mounting_angle_param_conf.angle_vertical = 0  
2mounting_angle_param_conf.enable = 1  
2mounting_angle_param_conf.angle_vertical = 50
```

## 3.19 设置/获取点云数据使能状态

深度数据转换为点云数据在内部已实现，如需获取点云数据仅需设置点云使能即可；点云数据将保存在 MeasureData 中的 point\_cloud\_data 中；

### 3.19.1 示例代码

```
HPS3D_SetPointCloudEn(true);  
HPS3D_GetPointCloudEn();
```

### 3.19.2 点云数据获取说明

获取点云数据前需要将光学参数补偿开启并且需要在初始化完成之后设置，

```
HPS3D_ConfigInit(&handle);  
.....  
HPS3D_SetOpticalEnable(&handle, true);  
HPS3D_SetPointCloudEn(true);  
连续测量时，点云数据保存在观察者回调函数 UserFunc(handle, event) 中的 event 事件中的  
MeasureData.point_cloud_data 中；单次测量时，点云数据保存在  
handle.MeasureData.point_cloud_data 中；
```

## 3.20 将测量输出无效值设置为指定特殊值

### 3.20.1 示例代码

```
/*将特殊测量输出值转换为指定特殊值参数配置*/  
HPS3D_ConfigSpecialMeasurementValue (true, 15000);
```

说明：

默认输出无效数据包含 65300, 65400, 65500, 65530 等值，这些值表明该像素点测量返回值无效，为方便用户统一处理，可调用该接口函数将这类特殊值设定为指定的数值进行统一处理；如上将所有无效值设置为 15000mm；

## 3.21 设置边缘噪声滤除

受设备角分辨率影响，可能存在物体边缘像素点光斑部分照射在物体本身，一部分照射在背景，从而导致该像素点返回的测量值为不稳定状态，从而形成被测物体与背景之前存在过度数据；设置该使能后对目标物体边缘像素点有明显的滤除和改善作用，但无法完全滤除；

此功能仅需设置为开启或关闭即可参数使用内部默认参数无需进行修改；

### 3.21.1 示例代码

```
HPS3D_SetEdgeDetectionEnable (true);  
HPS3D_GetEdgeDetectionEnable ();  
HPS3D_SetEdgeDetectionValue (1000);  
HPS3D_GetEdgeDetectionValue ();
```

## 3.22 保存点云数据为 ply 格式文件

### 3.22.1 示例代码

```
/*保存点云数据为 ply 格式文件*/  
HPS3D_SavePlyFile("pointCloud.ply", handle.Measure.point_cloud_data);
```

将输出的点云数据保存到文件中，ply 文件格式可使用 Meshlab 工具查看；

## 3.23 以太网版本连接服务器参数配置

设备默认 IP 地址为 192.168.0.10，端口号为 12345，子网掩码默认为 255.255.255.0，网关默认为 192.168.0.1；在进行连接前需要先输入设备 IP 及端口号，如 `HPS3D_SetEthernetServerInfo (&handle, "192.168.0.10", 12345);`；如需修改设备默认 IP 地址、端口号、子网掩码或网关需要调用重置服务器接口 `HPS3D_ConfigEthernet (...)`；如需永久生效还需保存通信配置；

### 3.23.1 示例代码

```
HPS3D_SetEthernetServerInfo (&handle, "192.168.0.10", 12345); /*子网掩码默认为  
255.255.255.0，网关默认为 192.168.0.1*/
```

重置服务器 IP 示例代码：

```
/*重置服务器 IP*/ /*传感器默认 ip 为 192.168.0.10 端口 12345，修改需要调用该函数*/  
uint8_t serverIP[4] = {192, 168, 0, 10};  
uint8_t netMask[4] = {255, 255, 255, 0};  
uint8_t gateway[4] = {192, 168, 0, 1};  
HPS3D_ConfigEthernet (&handle, serverIP, 12345, netmask, gateway);
```

注意：

重置服务器 IP 后，如需永久保存需要保存通信配置(参考 3.26)，保存后请牢记修改后的 IP，否则将无法再次连接；

## 3.24 自动连接设备并进行初始化配置

该接口将自动扫描当前环境中可连接的设备并进行初始化配置，返回值为当前连接成功的设备数量；

### 3.24.1 示例代码

```
HPS3D_HandleTypeDef handle[10];  
uint8_t connect_number = 0;  
connect_number = HPS3D_AutoConnectAndInitConfigDevice(handle);  
/*返回连接成功的设备数量*/
```

注：调用该接口后则不需要调用设备初始化接口；

### 3.25 设置/获取测量数据包类型

#### 3.25.1 示例代码

```
HPS3D_SetMeasurePacketType (DEPTH_DATA_PACKET);  
  
HPS3D_GetMeasurePacketType ();
```

注：该接口必须在设备连接前调用，若不设置则默认为深度数据输出；

### 3.26 保存通信配置

#### 3.26.1 示例代码

```
/*将当前通信参数保存为默认通信参数，并永久生效*/  
HPS3D_SaveTransportConf (handle);
```

注：调用此函数后，通信参数将不能恢复默认值，请牢记当前参数(请慎用)

### 3.27 设置/获取多机编码

多机编码是为了解决多设备协同工作时出现光信号之间产生的干扰现象，调用此接口设置各设备之间的多机编码便可将此干扰滤除，目前设备支持多机编码范围是 0~15；如需永久生效此配置需要在配置完成后调用 3.26 保存通信配置接口，或使用客户端软件(HPS3D\_Client.exe)进行配置；

#### 3.27.1 示例代码

```
HPS3D_SetMultiCameraCode (handle,0);  
UInt8_t cameraCode = HPS3D_GetMultiCameraCode(handle);
```

## 四、多设备支持

SDK 版本为 V1.7.11 及以上版本已支持多台设备连接；多设备连接将使用到 api.h 中的 DEV\_NUM, OBSERVER\_NUM 等相关宏定义，用户可修改这些宏定义的值以满足实际应用过程中的需求，修改其参数值将会影响到 SDK 内存的占用；

多设备连接主要接口函数为 **HPS3D\_AutoConnectAndInitConfigDevice()**；详细请参考 3.24 章节；注册事件 HPS3D\_AddObserver() 中的 callBack 函数地址可以相同也可以不同，相同时表示所有设备有数据返回均执行该回调函数，此时可通过返回的 ObserverID 进行区分；用户可根据需求进行设定，详细可参考相应的 Demo 程序；



## 五、常见问题

### 4.1 编码格式不匹配？

我司提供的 SDK 编码格式为 UTF-8;如在使用该 SDK 提供的 API 时出现编码格式错误,请在工程项目中新建 api.h 文件,并将本 SDK 提供的 api.h 内容拷贝至新建 api.h 文件中,即可解决编译错误;

### 4.2 如何获取点云数据和深度数据？

连续测量命令返回数据的方式是采用异步通知的方式;在连续测量时需要添加观察者具体操作请参看相应的 Demo 程序。添加观察者需要将回调函数的函数地址传入给观察者,该回调函数主要是对连续测量返回数据包进行处理;

点云数据配置需要在初始化完成之后执行(参考 3.19 章节)进行配置,点云数据在相应的 MeasureData 中可以得到,存储方式是按像素点顺序依次存放在数组中;

## 六、修订历史纪录

Date	Revision	Description
2018/12/11	1.0	初始版本。
2019/03/06	1.1	加入以太网版本配置及边缘滤除算法
2019/03/22	1.2	加入自动连接设备接口，支持多设备连接

HYPERSEN

**IMPORTANT NOTICE – PLEASE READ CAREFULLY**

Hypersen Technologies Co., Ltd. reserve the right to make changes, corrections, enhancements, modifications, and improvements to Hypersen products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on Hypersen products before placing orders. Hypersen products are sold pursuant to Hypersen's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of Hypersen products and Hypersen assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by Hypersen herein.

Resale of Hypersen products with provisions different from the information set forth herein shall void any warranty granted by Hypersen for such product.

Hypersen and the Hypersen logo are trademarks of Hypersen. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2018 Hypersen Technologies Co., Ltd. – All rights reserved